



NUI Galway
OÉ Gaillimh

Final Year Project Report

Port Knocking Protocol

Project Code - MS3

Cathal Corbett

16321973

Project Supervisor – Michael Schukat

Table of Contents

Table of Figures	4
Table of Tables	5
Acknowledgements	5
Chapter 1 – Introduction	6
Project Scope Outline	6
Outline Thesis Structure	6
Chapter 2 – Literature Review	7
Basic Port Knocking Implementation.....	7
Port Knocking in Industry	8
Port Knocking Security Concerns.....	8
Sniffing	8
Spoofing	8
NAT Attack.....	8
DOS (Denial of Service) Attack	8
Replay Attack.....	9
Out-of-Order Packet Delivery.....	9
Authentication vs Connection	9
Security Through Obscurity	9
Chapter 3 – Technical Review	10
Chapter 4 – Proposed Port Knocking Protocol.....	13
Proposed Port Knocking Protocol Outline	13
Port Knocking Protocol Title.....	13
Prerequisites.....	13
Client-Side Operation.....	13
Server-Side Operation.....	14
Software Tools.....	15
Protocol Programming Tools.....	15
Protocol OS Testing Environment.....	16
Protocol Design Decisions.....	17
Transport Layer Protocol Decision	17
Single vs Multiple Port Knocking Packets Decision.....	18
Encryption & Decryption.....	18
Asymmetric Cryptography	18
Hybrid Encryption.....	19

Digital Signature	21
Encrypt-Then-Sign or Sign-Then-Encrypt	22
Java Encryption Implementation	24
Final Decryption Implementation	25
Network Time Protocol Clock Synchronization	27
Server (Ubuntu 16.04) NTP Clock Synchronization	27
Client (Windows 10) NTP Clock Synchronization	31
Packet Capture	32
Client-Server Connection	36
Graphical User Interface	46
Port Knocking Server via Command Line	47
Port Knocking Client via Command Line	47
Chapter 5 – Testing.....	48
Correct & Fragmented Port Knocking Sequence.....	48
Incorrect Port Knocking Sequence	50
Late Delivery Packet Port Knocking Sequence	51
Time Modification Port Knocking Sequence	52
Without Encryption Port Knocking Sequence	53
Incorrect or Tampered Digital Signature Port Knocking Sequence	53
Falsely Prepend Client/Server ID in Packet Data	54
Client Public Key for Signature Verification Not Shared	54
Encryption & Decryption Testing	55
Port Scan Test.....	56
Chapter 6 – Conclusion.....	58
Future Work	59
References	60

Table of Figures

Figure 1 – Basic Port Knocking.	8
Figure 2 – Client-side implementation of the Port Knocking Protocol.	14
Figure 3 – Server-side implementation of the Port Knocking Protocol (does not show connection attempt).	15
Figure 4 – Sample Screenshot of master and feature Git branches.	15
Figure 5 – Screenshot of Ubuntu VM host-only network adaptor.	16
Figure 6 – Screenshot of Ubuntu VM NAT adaptor.	16
Figure 7 – Screenshot of Windows 10 host machine (client) IP Address.	17
Figure 8 – Screenshot of Ubuntu VM (server) IP Address.	17
Figure 9 – TCP three-way handshake.	17
Figure 10 – Symmetric vs Asymmetric Encryption.	19
Figure 11 – Hybrid Encryption procedure.	21
Figure 12 – Digital Signature procedure.	22
Figure 13 – NTP servers that are publicly available for Ireland.	27
Figure 14 – Sample screenshot of multiple static public IP that exist for one Ireland NTP server.	28
Figure 15 – NTP server for National University of Ireland, Galway.	28
Figure 16 – The 'ntp.conf' file that contains and loads the NTP server for application use.	29
Figure 17 – Shows iptables config file to load firewall commands on VM start-up for NTP clock synchronization.	30
Figure 18 – Firewall iptables rules loaded for NTP clock synchronization.	30
Figure 19 – Start NTP service and show active status.	30
Figure 20 – The 'ntpq' output showing running NTP servers.	31
Figure 21 – The 'timedatectl' showing that the RTC clock is running NTP clock synchronization.	31
Figure 22 – Setting up the host machine (Windows 10) with an NTP server.	31
Figure 23 – Setting RTC clock of host machine (Windows 10) to using NTP clock synchronization. ...	32
Figure 24 – Sample TcpDump output of UDP packets for specific server ports.	33
Figure 25 – Sample iptables firewall rules for client-server connection.	37
Figure 26 – Unsuccessful telnet attempt to Port Knocking server port 23.	38
Figure 27 – Wireshark capture of unsuccessful telnet attempt.	38
Figure 28 – Successful telnet attempt to Port Knocking server on connection port 3000.	38
Figure 29 – Wireshark capture of successful connection attempt on connection port 3000.	39
Figure 30 – Port Scan of telnet port 23 and connection port 3000.	39
Figure 31 – Running the Port Knocking server via terminal.	47
Figure 32 – Running the Port Knocking client via command prompt.	47
Figure 33 – Port Knocking daemon log output of successful Port knocking connection.	49
Figure 34 – Server iptables firewall rules configured for single connection knock.	49
Figure 35 – Port Scan of server port 23, currently configured connection port and previously configured connection port.	50
Figure 36 – Port Knocking daemon log output of unsuccessful Port knocking connection followed by successful attempt.	51
Figure 37 – Port Knocking daemon log output of UDP late packet arrival.	52
Figure 38 – Port Knocking daemon log output of time modification of UDP packet.	53
Figure 39 – Port Knocking daemon log output of unencrypted Port knocking sequence.	53
Figure 40 – Port Knocking daemon log output of Port Knocking attempt of tampered with digital signature.	54

Figure 41 – Port Knocking daemon log output of falsely appended client/server IP address.	54
Figure 42 – Port Knocking daemon log output when the server contains no associated client public key.	55
Figure 43 – Graph of Hybrid vs Asymmetric Packet Encryption Time.	56
Figure 44 – Graph of Hybrid vs Asymmetric Decryption Time.	56
Figure 45 – Time taken to port scan all TCP server ports.	57
Figure 46 – Time taken to port scan all TCP and UDP server ports.	57

Table of Tables

Table 1 - Strengths and weaknesses of previously implemented Port Knocking protocols.	11
Table 2 – Client & Server requirements for successful Port Knocking connection.	13
Table 3 – UDP vs TCP packet transmission between client-server for Port Knocking connection.	18
Table 4 – Number of packets to be sent to brute-force a Port Knocking server.	18
Table 5 – Comparison of RSA and AES encryption and decryption times.	20
Table 6 – TcpDump flag descriptions.	33
Table 7 – Description of Port Scan ‘STATE’ fields.	40
Table 8 – Description of above daemon log output of client Port Knocking sequence.	51
Table 9 – Test results of Hybrid vs Asymmetric encryption and decryption.	55

Acknowledgements

I would like to express my very great appreciation to Dr Michael Schukat for continually taking the time to meet with me throughout this project and provide valuable insights and guidance.

Lastly, I must give my sincere gratitude to my friends and family for their unending support and encouragement throughout my university career.

Chapter 1 – Introduction

Project Scope Outline

In computer networking, Port Knocking is a method of externally opening ports on a firewall by generating a connection attempt on a set of pre-specified closed ports. Once a correct sequence of connection attempts is received, the firewall rules are dynamically modified to allow the host which sent the connection attempts to connect over specific port(s). A variant called single packet authorization exists, where only a single 'knock' is needed, consisting of an encrypted packet. The primary purpose of Port Knocking is to prevent an attacker from scanning a system for potentially exploitable services by doing a port scan, because unless the attacker sends the correct knock sequence, the protected ports will appear closed. In this project I will design, implement and validate a Port Knocking Protocol that is suitable for IoT applications.

The protocol designed should take into consideration the computing and processing constraints due to limited hardware that exist with the majority of IoT devices. Designing a security protocol for such devices should not negatively affect the device's intended operations or overly pressurise the device's hardware resources such as CPU's, memory, storage etc.

A security protocol can nearly be guaranteed to use some cryptographic standards or algorithms. Encrypting internet traffic in the present time is a necessary standard due to the amount of packet eavesdropping, modification and fabrication that occurs from malicious attackers who will use all attempts to compromise a service. However, unlike modern computers IoT devices cannot compute as efficiently, encryption and decryption functions, which utilise valuable hardware such as processors and storage. This will be investigated within the thesis.

Because IoT devices are subject to and vulnerable to attackers that perform port scanning, DOS attacks and other malicious attacks to compromise a service, I intend to find a way to protect the port of the IoT device that is running these valuable services to make it more difficult for an attacker to find what port is accepting connections.

As a final deliverable, I expect to have developed a secure, resource efficient and computationally time adequate Port Knocking Protocol that suits the operations of IoT devices.

Thesis Structure

Chapter 2 of the report is an introduction and discussion on the basic implementation of the security protocol as a form of literature review and its use in both academic and industry contexts.

Chapter 3 broadens the review technically by discussing how the protocol has been modified and improved since its first implementation discussing techniques and methodologies used and the resulting strengths and vulnerabilities of all improvement made.

Chapter 4 examines the implemented Port Knocking Protocol and discusses decisions made and technical issues experienced during the project.

Details of project results, evaluation and final deliverables are given in Chapter 5.

Finally, the project will be reviewed in Chapter 6 in terms of a conclusion and future work considered.

Chapter 2 – Literature Review

The Port Knocking Protocol has been a research topic for most of the past twenty years. The original implementation, although extremely unsecure, was proposed as an extra layer of security allowing users to protect their devices from untrustworthy internet users.

Prior to this proposed protocol the 'firewall' was the only mechanism that existed but will only accept network traffic from client IP addresses manually configured in the firewall rules. The firewall can be compromised through IP spoofing where an undesired attacker will modify the source IP address of a network packet which is accepted by the firewall rules and may easily gain access to the network.

A huge proportion of servers have either no or extremely weak password protection beyond the firewall on their devices making it even easier to access a device. IoT devices such as cheap webcams are subject to this behaviour and often silent attacks might never be detected by the device owner as the system is rarely monitored by the user or the device might not recognise an undesired attacker.

Therefore, an approach should be taken that involves the layering of security protocols along with the firewall to protect a device. Port Knocking is one protocol which is proposed to contribute to this layering of security on a device or server. The main aim of Port Knocking as a security protocol is to authenticate a user wishing to gain access to a port and to allow firewall rules to be amended dynamically to allow authenticated traffic access to the server port.

Basic Port Knocking Implementation

The original Port Knocking Protocol is a simple concept of externally opening ports on a firewall by generating a sequence of connection attempts on a set of prespecified closed ports [1]. Both the client and server have an agreed 'secret' which is a sequence of port numbers representing a single server port number which a client wants access to.

The client forms network packets (usually TCP or UDP) with a port number from the specific port sequence contained in each packet. The client makes a connection attempt to the server's closed ports. At the server side a Port Knocking daemon is monitoring the log files of the firewall or uses a packet capture tool. If the daemon recognises the sequence of packets based on the 'secret', the Port Knocking daemon will externally modify the firewall rules allowing a connection from the source IP address in the client packets to the server port. Essentially, the server port will become open to the client. If a knocking sequence is not recognised the daemon will ignore the sent packets. The only indication to an outside source a connection attempt has occurred is at the end of the knock sequence by doing a port scan to check if the port expected is opened or closed.

[2] shows the simplest implementation of the Port Knocking Protocol. The client and server have a shared secret agreement of the Port Knocking sequence – [1145, 1087, 1172, 1244, 1031] representing the server port 22 (SSH) which the client wishes to get access to. The client firstly sends this sequence which is recognised by the server daemon who opens port 22 to the client IP to initiate an SSH session.



Figure 1 – Basic Port Knocking. [2]

Port Knocking in Industry

Due to security concerns discussed below, Port Knocking has been originally slow to evolve in industry and used to protect businesses computer systems. As the protocol has been modified and better supported, the protocol has been since been deployed by many businesses to protect their systems. Majority have reported a significant reduction in the amount of bandwidth consumed by things like SSH brute-force attacks as a result.

Port Knocking Security Concerns

Major faults exist with the basic implementation of the Port Knocking Protocol as discussed below.

Sniffing

An undesired attacker can monitor the network using a tool such as Wireshark and easily extract the Port Knocking sequence if unencrypted. Therefore, this predefined 'secret' could become compromised and easily used by an attacker to gain access to the server.

Spoofing

IP spoofing is the creation of IP packets which have a modified source address in order to either hide the identity of the sender. An attacker can change the source IP address of packets pretending to be a desired user. This is dangerous if the attacker has previously sniffed out the port sequence though use of a packet capture and possible packet decryption and intends to get access to the server pretending to be another client. [3]

NAT Attack

NAT attacks occur when multiple devices on a network share one or limited public IP address. This has occurred due to the IPv4 address exhaustion. When a packet is sent from a client and reaches the router, the IP address will be replaced with the routers assigned public IP address which is sent out externally. If a port sequence sent out has gone through the NAT procedure client-side and the server allows access to that public IP address, any device on that client-LAN network can connect to the client and the server has no way of verifying that the correct device has access. [4] [2]

DOS (Denial of Service) Attack

Occur when an attacker overloads the server by hitting it with multiple invalid network packets while valid traffic in return may not make it to the server. If the server is known to be vulnerable this can

have serious consequences slowing it down or worst case, causing it to fall over. A brute-force attack is a type of DOS attack where an attacker continually hits a sequence of random closed ports and performs a port scan to check for any port activity changes with the intention of getting access to the server. [4] [3]

Replay Attack

Type network attack where valid data is transmitted maliciously or fraudulently repeated or delayed. In terms of Port Knocking, an attacker may try to replay a sniffed Port Knocking sequence using their own IP address and gain unauthorized access to a server. [4]

Out-of-Order Packet Delivery

Applies to UDP packets as they were designed for speed and not reliability and the protocol does not even guarantee successful delivery of the UDP packet. Packets sent out after one another can become out-of-order if they take different network paths. This can occur if part of the network experiences downtime or uptime in the middle of sending out a Port Knocking sequence, the shortest path needs to be recalculated and can cause the packets to become disordered. Therefore, if the server receives one packet of the packet sequence out of order from the client the Port Knocking attempt is redundant. Often, if the implemented protocol is configured to use UDP packets and an unsuccessful attempt occurs due to out-of-order delivery, the client will try to resend the sequence after some timeout. [3] [2]

Authentication vs Connection

All Port Knocking protocols developed identify a Port Knocking sequence from a client and allow a client-server connection. However, very few protocols have tried to make sure the client is who they say they are and not an undesired attacker. Authentication proves that the client sending the Port Knocking sequence is actually them and the Port Knocking sequence packets have not been modified or spoofed in some way by an undesired attacker. [2]

Security Through Obscurity

Is the reliance in security engineering (whether it be a security protocol or even some encryption standard), on design or implementation secrecy as the main method of providing security to a system [5]. The basic implementation of the protocol relies heavily on the concept of 'Security Through Obscurity'. For a client-server connection to occur, the protocol confides only on a set of pre-specified closed ports between both machines.

However, the Port Knocking Protocol has evolved as a form of authentication. In reality, a machine can never be 100% secure using one security standard or even multiple standards. However, the idea is to take multiple security mechanisms and layer them to form a robust security system that will make it as difficult as possible for an attacker to gain access to a system. Although the basic implementation of the protocol, if solely implemented to guard a system, would be coined the term 'Security Through Obscurity'. Instead, using the Port Knocking Protocol along with other security standards (firewall, encryption, digital signature, digital certificates, password protected ports, IPsec to name a few) forms a security system that cannot be said to rely on 'Security Through Obscurity'.

Chapter 3 – Technical Review

This chapter expands the literature review and discusses technically, the advances the Port Knocking Protocol. Protocols developed have adapted by incorporating established security techniques and standards to build upon the basic implementation of the protocol which reduces or eliminates some of the security concerns that have been previously considered in Chapter 1. However, not all protocols address the above concerns, depending on their use cases which will be revealed below.

The original protocol has been advanced throughout the years improving upon the above security concerns. Such improvements include encryption of data packet contents, single-packet knocking, single-packet authentication, use of IPsec to create a Virtual Private Network (VPN) tunnel, use of steganography, hashing to spoof client IP addresses in the Port Knocking sequence, timestamp synchronisation to address the out-of-order delivery and the by-passing of the firewall by opening a port to all network traffic where the Port Knocking daemon would monitor traffic for the Port Knocking sequence.

The security of the protocol using the above techniques has improved immensely but can still be compromised. Many of the protocols implemented use a large amount of compute power to ensure a secure and authenticated connection between the client and server.

The below table [6] shows the improvement of Port Knocking from its first existence up to about 2012.

Port Knocking Project	Summary	Strength	Weakness
Basic Port Knocking	First introduction of Port Knocking concept	Firewall rules used to dynamically open/close a port	Replay packet attack. Scanning. Out of order packet delivery
Advanced Port Knocking Suite	DES used in packet sequence	First use of encryption with knocking packet	Could be slow due to encryption/decryption
Barricade	ICMP echo request as knocking packet	Simple to implement	Password in ICMP packet can be sniffed
Cryptknock	Encrypt knocking sequence in one packet	Difficult to replay the knocking packet by sniffing with TCP dump	Data read by libpcap where it configures under monitoring state
Doorman	Single UDP packet knocking	Uses MD5 hash	Rainbow table can crack MCD5 hash table
Knockd	Combine UDP & TCP packet knocking	Able to use UDP & TCP packets	Same implementation of basic port knocking
Sig2Knock	Randomize the port sequence	Overcomes port scanning	Difficult to implement
Pasmal	Encrypted ion packed for TCP and ICMP	Able to use ICMP & TCP packets	Encryption may slow down performance
Portkey	TCP ports from 1 - 65535 can be used	Many TCP ports can be used for knocking	Only supports IP table-based firewall
Cryptography of Knocking [3]	Port knocking with cryptography	More secure than basic port knocking	Only supports IP table-based firewall
Cerberus by Dana Epp	ICMP packet sent to knocking server	Applied special ICMP ping packet	Only supports IP table-based firewall

Port Knocking with Single Packet Authorization [3]	Single packet used as an authentication mechanism	Authentication packet encrypted & difficult to replay	Out of order packet delivery is not discussed
One Time Knocking Framework using SPA and IPSec [7]	Enhanced SPA by using IP Sec	Knocking password sent to smartphone by RNG server	IPSec with firewall rules is difficult. Complex system.
Network Security using Hybrid Port Knocking [8]	Combination of cryptography, steganography & mutual authentication	Difficult to replay	Overhead increased due to cryptography & steganography
Advanced PK Authentication Scheme with QRC using AES [9]	QRC spoofs IP address (QRC = Quadratic Residue Cipher)	Port scans difficult. Hard to replicate IP address.	The complexity of the protocol may slow performance.

Table 1 - Strengths and weaknesses of previously implemented Port Knocking protocols.

In most recent times there have been papers proposed to further improve the protocol.

Covert Communication Using Port Knocking [10] proposes a new covert channel for stealthy communication. The channel uses Least Significant Bit (LSB) steganography and Tariq Port Knocking to hide data. Tariq Port Knocking was first proposed in Network Security using Hybrid Port Knocking [8] which used both steganography and encryption. Additionally, GNU Privacy Guard (GnuPG) encryption is applied before hiding the data in the packet to add another layer of protection. Using Peak Signal to Noise Ratio the communication efficiency has been tested and the channel can achieve 152 bps as a maximum transmission rate. Despite the increased overhead due to steganography and encryption the protocol defends against DOS attacks, spoofed packets and TCP replay attacks.

The scalableKnock (sKnock) protocol [4] is a one-way authentication protocol that requires the client to send a UDP authentication packet before opening a connection to the server behind a firewall. The packet contains the client's certificate and the port number of the server it requires connection to. The privacy of the client is protected by asymmetrically encrypting packet contents with a freshly chosen ephemeral key derived from the server's public key using Elliptic-curve Diffie-Hellman (ECDH) encryption. To reduce the number of packets involved in the authentication to just one UDP packet, Elliptic-curve cryptography (ECC) public keys are used which are lengths of up to 256 bits. This results in a packet size of 800 bytes keeping within the network maximum transmission unit (MTU) size of 1500 bytes. RSA public keys couldn't be used due to lengths of 2048 bits which would have exceeded the MTU size. This MTU size will have to be taken into consideration in the proposed protocol.

Another proposed Port Knocking protocol uses Network Time Protocol (NTP) synchronization [3]. A knock sequence is less venerable to replay and brute force attacks if its lifespan is shorter. Therefore, the client and server share the same time by both sending a synchronization request at knock daemon start-up. The protocol states that the NTP 'time' function only gives granularity of a second which might not be good enough when we take into account the how fast a network can transfer packets. Upon further research I have found that NTP can now achieve granularity of tens of a millisecond over public internet and becomes even more efficient in a Local Area Network (LAN) [11]. Clock skew can occur due to asymmetric routing and network congestion, so it is important to implement repeated NTP queries to synchronize the client and server every so often.

The simple port knocking method [6] proposes to remove the added complexity of integrating a firewall and instead leave open a random listening port where the Port Knocking sequence can be sent over an Secure Shell (SSH) connection allowing the client to connect straight to the server. The port number from SSH is predefined and the common port 22 will not be used. The proposed method has a very simple implementation while defending against replay attacks and port scanning.

Certificate-Based Port Knocking [12] allows both client and server to hold a digital certificate by some certificate authority. Using this type of asymmetric encryption, the client's identity is authenticated even before the communication begins. This addresses the problem of the shared Port Knocking secret to be invalidated at the firewall in conventional Port Knocking methods.

Secure Port Knock-Tunnelling [13] protects against NAT and DOS attacks. DOS attacking is solved when a Port Knocking sequence is successfully sent the firewall opens one port for the client and triggers a VPN connection. However, in the sending of the knocking sequence stage UDP packets are sent. To avoid the problem of out of order delivery a UDP packet is only sent every 10 seconds. Therefore, the process of four knocks will take at least 40 seconds with the buffer flushing automatically if the packets do not arrive in time. Having to wait this amount of time for a client-server to occur makes the protocol ineffective. Incorporating a timeout on the client side I believe would be a better solution if it happened that packets became out of order or lost.

Chapter 4 – Proposed Port Knocking Protocol

Chapter 3 introduces a rough outline of the designed protocol. Further examination takes place into software and design decisions, technical issues encountered and overall protocol implementation.

Proposed Port Knocking Protocol Outline

Port Knocking Protocol Title

N UDP packet sequence Port Knocking Protocol using ‘Encrypt-Then-Sign’ RSA-AES Hybrid Encryption and Digital Signature with client-server connection of N random ports.

Prerequisites

- Client and server have their own asymmetric keys which are generated and distributed by some means –
 - C_{PUB} and C_{PRIV} → indicate client public and private keys.
 - S_{PUB} and S_{PRIV} → indicate server public and private keys.
- Both C_{PRIV} & S_{PRIV} are secretly kept by both identities.
- The server and client exchange public keys (C_{PUB} & S_{PUB}).
- Both client and server have agreed on a Port Knocking sequence (K_0, K_1, \dots, K_{N-1}) where N is the sequence size.
- The destination port number (P_{NUM}) where the server is running the required service the client wants access, needs not to be shared with the client and can be protected.
- The server stores each client IP address along with their C_{PUB} which is used for digital signature verification purposes.

Client Requirements	Server Requirements
1. C_{PRIV}	1. C_{PUB}
2. S_{PUB}	2. S_{PRIV}
3. K_0, K_1, \dots, K_{N-1}	3. K_0, K_1, \dots, K_{N-1}
4. Client & Server IP Address	4. P_{NUM}
	5. Client & Server IP Address

Table 2 – Client & Server requirements for successful Port Knocking connection.

Client-Side Operation

- Depending value of N the client will generate N random numbers between 0 and 65535 which will be used as connections ports (C_0, C_1, \dots, C_{N-1}).
- Client generates N 256-bit symmetric AES key ($AES_0, AES_1, \dots, AES_{N-1}$) for each data packet ($DP_0, DP_1, \dots, DP_{N-1}$).
- Data strings (DS_i) are formed containing the NTP timestamp, C_i , server IP address and client IP address.
- Each DS_i is encrypted using the AES key (AES_i) forming an encrypted data string (DS_i^e).
- AES_i is encrypted with S_{PUB} using RSA encryption algorithm forming an encrypted AES key (AES_i^e).
- DP_i is a string containing DS_i^e and AES_i^e .
- Using C_{PRIV} , the DP_i is hashed and signed using the Java ‘SHA with RSA’ algorithm forming the signed data packet (DP_i^s).

- Both DP_i and DP_i^S are appended to a UDP packet and sent to the server using the server IP address and K_i as the packet port number.

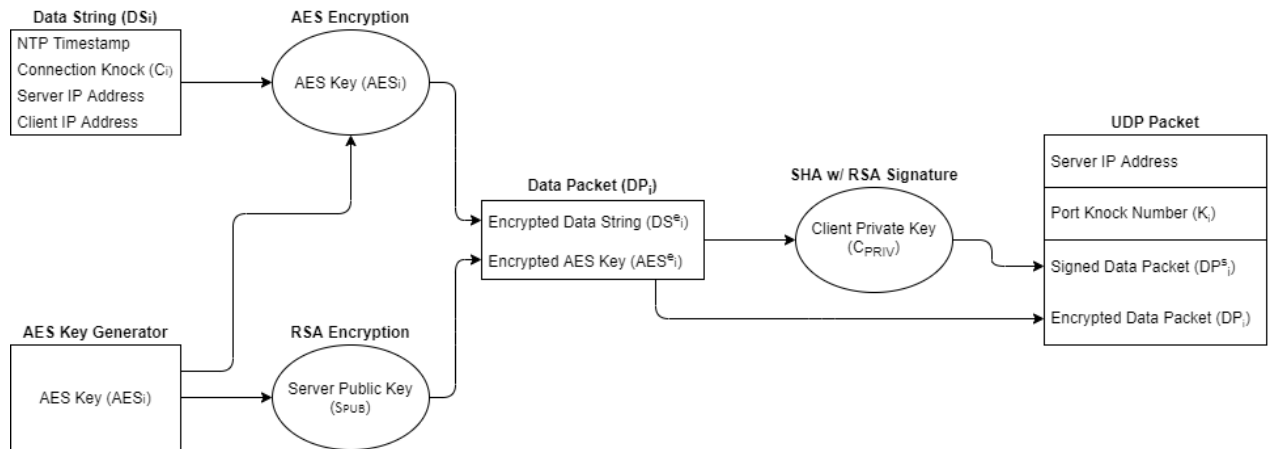


Figure 2 – Client-side implementation of the Port Knocking Protocol.

Server-Side Operation

- Sequence of UDP packets arrive server-side.
- The daemon verifies each UDP packet by verifying the signature hash. Computing the hash on DP_i and RSA signature decrypting DP_i^S using C_{PUB} can verify the signature attached to the packet.
- Once verified, the DS_i^e and AES_i^e are retrieved from the DP_i .
- The AES_i^e is decrypted using S_{PRIV} to produce AES_i .
- DS_i^e is then decrypted using the AES_i to produce the original plaintext data string DS_i .
- From DS_i the following plaintext information is retrieved – NTP timestamp, C_i , server IP address and client IP address.
- Both the server IP and client IP addresses are verified to make sure of no packet tampering/modification.
- Once the correct Port Knocking sequence has been identified by the daemon a buffer of connections ports (C_0, C_1, \dots, C_{N-1}) should be obtained.
- The daemon will now amend the firewall rules of the server allowing a client-server connection though C_0 to the server for a specific amount of time and then the firewall rules are amended to close the port and open C_1 and so on until C_{N-1} is reached and the connection fully closes.
- However, what is happening is that for each time a connection port is opened the firewall is routing packets from that port (C_i) to P_{NUM} that contains the service that the client wishes to get access to.

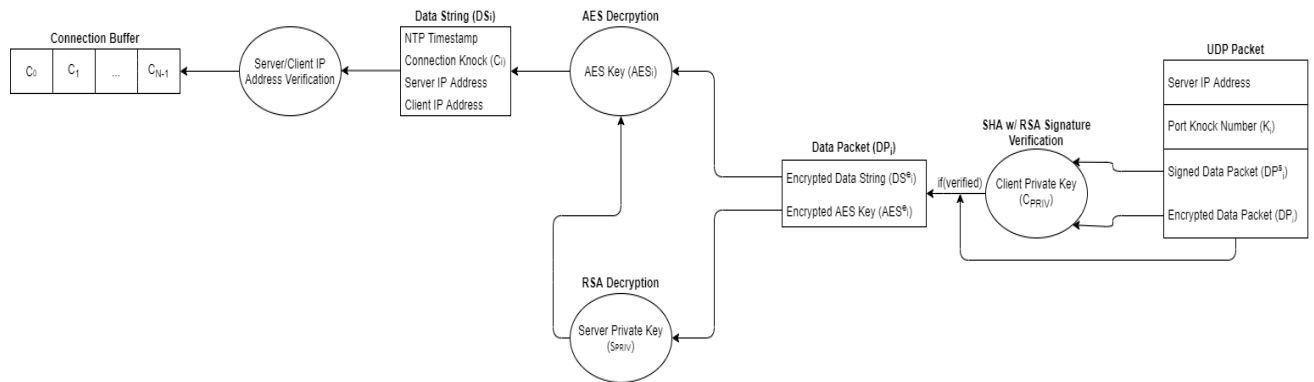


Figure 3 – Server-side implementation of the Port Knocking Protocol (does not show connection attempt).

Software Tools

Protocol Programming Tools

For the Port Knocking Protocol implementation, Java JDK 8 programming language is used. The choice of programming language depended on being able to apply network sockets which all C type programming languages can implement. Java was decided on because it is a well-established language with worldwide support and well documented APIs around sockets while many opensource Java libraries have been previously created for features that my protocol would potentially use.

The Java Eclipse IDE has been used for writing code and testing. Once Build Paths and Class Paths have been configured, Eclipse is an easy-to-use tool for Java programming as it supports all standard Java libraries, predictive text caters for easy coding and allows for in-depth debugging and JUnit 4 testing.

A distributed version control system is to host all Java code and manage the project. A GitHub repository was set up with master and feature branches. When features of the protocol have been successfully completed merges take place from the feature branch into the master branch allowing for correct and successful project management. The Eclipse IDE supports Git repository cloning, committing, branching, merging etc. allowing one to write and manage code from a single tool which proves useful when working on a project.

Default branch

master

 Updated 2 months ago by ccathal

Default

Change default branch

Your branches

server_gui

 Updated 13 hours ago by ccathal

0 | 15

New pull request

client_gui

 Updated 13 hours ago by ccathal

0 | 8

New pull request

encryption_digital_signature

 Updated 18 days ago by ccathal

0 | 11

New pull request

npt_time_synchronization

 Updated 22 days ago by ccathal

0 | 9

New pull request

packet_capture

 Updated last month by ccathal

0 | 8

New pull request

View more of your branches

Figure 4 – Sample Screenshot of master and feature Git branches.

Protocol OS Testing Environment

A Port Knocking Protocol requires a client-server connection. Such an environment needs to be set up to allow for the testing of the protocol. A Windows 10 OS running Eclipse IDE, intended to be the client, supported the sending of the Port Knocking sequence. Windows 10 ran Oracle VirtualBox where a Linux Ubuntu 16.04 OS image ran the Port Knocking daemon. For the testing environment to be successful the following conditions needed to be met –

- Both host (Windows 10) and guest (Ubuntu 16.04) needs to access the internet for GitHub project management.
- Host can ping guest and vice versa. Both host and guest needed a static IP address for a client server-connection to occur.

In the Ubuntu virtual machine setting two adaptors [14] were set up to solve the above problem –

- NAT adaptor allowed the virtual box access to an internet connection.
- Host-Only adaptor allowed a static IP address (192.168.56.101) to be assigned to the Ubuntu VM. The Windows 10 host will have 192.168.56.1 as an IP address for the internal network (VirtualBox Host-Only Network).

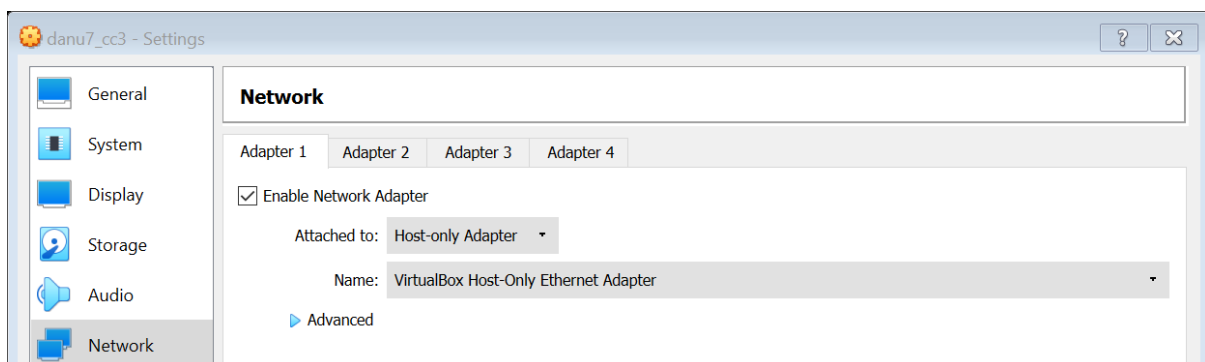


Figure 5 – Screenshot of Ubuntu VM host-only network adaptor.

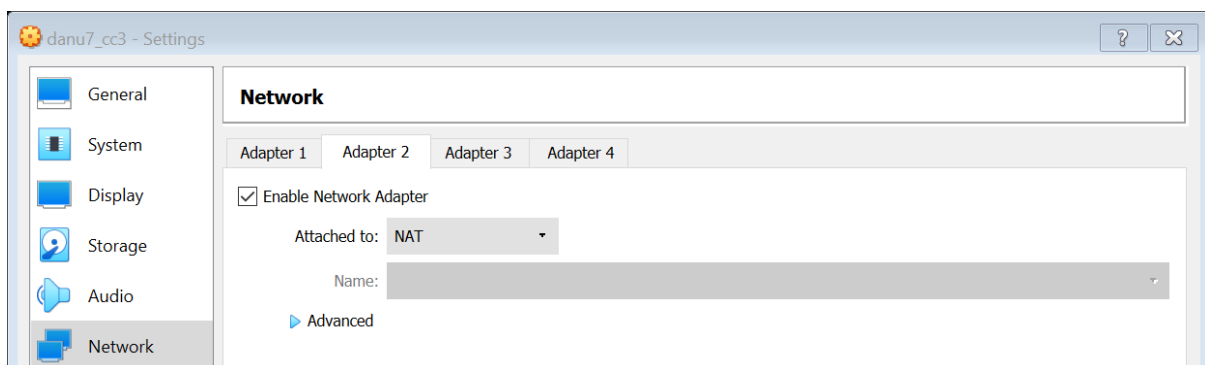


Figure 6 – Screenshot of Ubuntu VM NAT adaptor.


```
Command Prompt
C:\Users\catha>ipconfig

Windows IP Configuration

Ethernet adapter VirtualBox Host-Only Network:

    Connection-specific DNS Suffix  . : 
    Link-local IPv6 Address . . . . . : fe80::e0f0:9222:a9f:effc%7
    IPv4 Address. . . . . : 192.168.56.1
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . :
```

Figure 7 – Screenshot of Windows 10 host machine (client) IP Address.

```
cc3@cc3-VirtualBox: ~
cc3@cc3-VirtualBox:~$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:a7:ff:ac
        inet addr:192.168.56.101  Bcast:192.168.56.255  Mask:255.255.255.0
        inet6 addr: fe80::cccd:3dc0:2312:73dd/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:6 errors:0 dropped:0 overruns:0 frame:0
        TX packets:30 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:1499 (1.4 KB)  TX bytes:4025 (4.0 KB)
```

Figure 8 – Screenshot of Ubuntu VM (server) IP Address.

Protocol Design Decisions

Transport Layer Protocol Decision

It is decided that UDP (User Datagram Protocol) packets would be used. The conventional TCP (Transmission Control Protocol), although assures in-order packet delivery and retransmission of lost packets, requires the three-way handshake for data to be sent between client and server. Below shows the handshake required for a TCP connection.

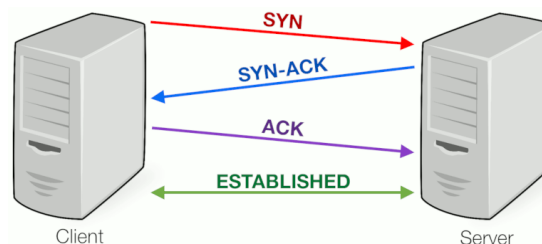


Figure 9 – TCP three-way handshake. [28]

A TCP protocol sends ACK (acknowledgement) packets back to the client after every successfully delivered packet. The three-way handshake and acknowledgement sending requirements significantly increase the time for a Port Knocking connection to occur. Due to the considerable amount of packet sending, and acknowledgements the benefit of Port Knocking being a stealth protocol is lost. If an attacker monitoring the network sees numerous connection attempts to a sequence of closed ports and acknowledgements it will give off an idea that the 'undetectable' security protocol may be monitoring the router for port sequences. Sending UDP packets does not require this initial handshake or acknowledgement packets reducing the time to make a Port Knocking connection. Additionally, UDP packets were designed for fast transmission. The protocol should only be compromised if an attacker identifies several UDP packets sent to a sequence of

closed ports. However, with a four-packet knock sequence the number of packet transmissions to achieve a client-server connection will be significantly reduced using UDP rather than TCP.

	Transmission Control Protocol	User Datagram Protocol
Initial Handshake	3 packs	-
Client Port Sequence	4 packets	4 packets
Server Acknowledgements	4 packets	-
Total Packet Transmission	11 packets	4 packets

Table 3 – UDP vs TCP packet transmission between client-server for Port Knocking connection.

It is evident that a TCP Port Knocking connection requires twice the amount of packet transmissions (after the handshake has occurred) compared to a UDP connection due to the server TCP acknowledgements. With the large volume of network traffic that exists on the internet nowadays identifying a UDP Port Knock sequence should be unlikely.

Single vs Multiple Port Knocking Packets Decision

The proposed protocol uses multiple packets instead of a single Port Knocking packet. If a single packet becomes intercepted and decrypted the whole message is compromised. Sending multiple packets reduces the chances of the whole message becoming established. If one packet in a multiple packet sequence is captured the whole message is not compromised.

Additionally, sending multiple packets reduces brute force attacks from undesired hackers. On a typical device with 65536 ports and a 3-packet Port Knocking sequence, an attacker must try every 3-port sequence in that port range and do a port scan between each attack to check if there has been a change in any port activity. The average cases are given below of the average number of packets to be sent by an attacker to successfully perform brute force on the server to gain access to the corresponding port of the Port Knocking sequence. Evidently, the average brute force attack packet number increases exponentially just by increasing the port sequence linearly.

Port Knock Sequence Length	Average Brute Force Packet Number
3-Packet Port Knock Sequence	$\frac{65536^3}{2} \approx 1.4 \times 10^{14}$
4-Packet Port Knock Sequence	$\frac{65536^4}{2} \approx 9.2 \times 10^{18}$
5-Packet Port Knock Sequence	$\frac{65536^5}{2} \approx 6.0 \times 10^{23}$
6-Packet Port Knock Sequence	$\frac{65536^6}{2} \approx 4.0 \times 10^{28}$

Table 4 – Number of packets to be sent to brute-force a Port Knocking server.

Encryption & Decryption

Asymmetric Cryptography

Many of the previous Port Knocking protocols implemented are not scalable. One reason for this is due to the use of symmetric cryptography where only one client and one server share the same public key. Client-side data contents are encrypted into the network packet using the public key and are decrypted server-side using the same public key. Firstly, the Port Knocking Protocol can only

scale to one server and one client using the above approach. If a server has shared the Port Knocking sequence to N clients, the number of public keys that need to be distributed is also N . If the public key becomes compromised an attacker can use the key to intercept and decrypt the Port Knocking sequence on the network allowing the attacker direct access the server through the port knocking sequence. Symmetric cryptography has also the added problems of secure key distribution and exchange which make it easy for undesired attackers to access the public keys through eavesdropping or some other means. The server also needs to manage all N public keys for each client efficiently which can prove troublesome.

Therefore, the protocol implemented intends to use asymmetric cryptography which allows multiple clients access one server using just one public key making a more scalable solution. Asymmetric encryption is based off using a public key which is publicly known and distributed and a private key which is known by only the server. The public key is used to encrypt the message client-side and server-side the message is decrypted using the private key it owns. Therefore, if a message becomes intercepted on the network by an attacker, the public key, which it may have obtained, will not be able decrypt the message as only the private key has such functionality.

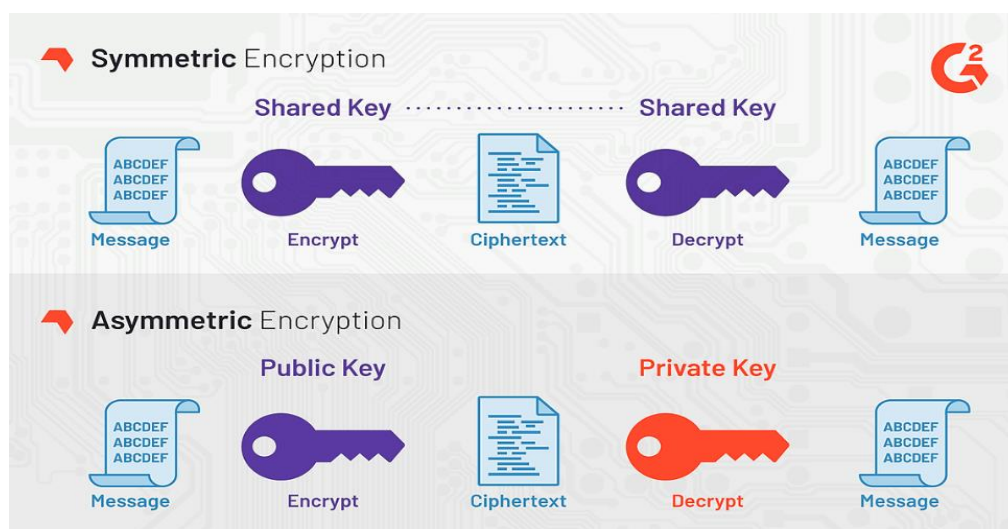


Figure 10 – Symmetric vs Asymmetric Encryption. [15]

Hybrid Encryption

The advantage of symmetric cryptography is speed. Generally, less CPU cycles are required for symmetric encryption and decryption. The below table [16] is from a survey conducted analysing the performance of AES against RSA encryption and decryption times. The AES algorithm outperforms the RSA comparison in both cases. It is also expected that as security of the asymmetric key increases to RSA 2048-bit alternative, the RSA compute time will worsen dramatically. Symmetric schemes are therefore orders of magnitude faster and less power hungry than an asymmetric scheme.

Algorithm	Packet Size (KB)	Encrypt Time (sec)	Decrypt Time (sec)
AES	118	1.7	1.2
RSA		10.0	5.0
AES	196	1.7	1.24
RSA		8.5	5.9

AES	868	2.0	1.2
RSA		8.2	5.1

Table 5 – Comparison of RSA and AES encryption and decryption times.

Hence, the Port Knocking protocol will implement a hybrid cryptosystem taking the advantages of both symmetric and asymmetric cryptography. The scalable solution of the asymmetric cryptosystem combined with the speed of the symmetric cryptosystem is used in the hope of producing a scalable protocol with faster encrypting and decrypting stages.

In a real-world scenario, the computational time of symmetric key generation and hybrid encryption at the client side is irrelevant as the Port Knocking sequence sent out will only be a handful of packets. What will be more important is designing a protocol that is secure from undesired attackers. Hence, the protocol proposes a symmetric session key will be produced by the client each time a knock packet is sent. The generation of a public key each time a packet is sent is the trade-off of computational time for producing a securer protocol. Also, this method provides forward secrecy, in that, if one message were broken in terms of the symmetric encryption, the adversary wouldn't be able to decrypt any of the other messages, because they all used different AES keys. The Port Knocking daemon should be robust, fast and capable of interpreting and decrypting a few hundred Port Knocking packets per minute which a hybrid cryptosystem intends to achieve.

Below shows the basic implementation of a hybrid cryptosystem. An asymmetric 'session' key is generated. The session key is used to encrypt the plaintext by applying it to a block cipher or hashing algorithm producing ciphertext. The servers asymmetric public key is then used to encrypt the session key producing an encrypted session key. Both the encrypted key and ciphertext generate the output message. This message is sent to the sever through some communication means (UDP connection, in our case) where hybrid decryption takes place. The encrypted session key is decrypted using the servers private key (which only they know) producing the original symmetric session key. The session key is used to decrypt the encrypted ciphertext producing the original plaintext message.

Again, the main benefit of the hybrid approach is to speed up the decryption process of the server which can be troublesome if the daemon is bombarded with incoming Port Knocking sequences or if an undesired attacker wishes to disrupt the Port Knocking service performing Denial of Service (DoS) attacks. Remember, if an attacker successfully disrupts the Port Knocking daemon through DoS attacks or some other undesired measures and the service experiences downtime, the server should

have other implemented security measures to protect itself through password protected services and the firewall which has been set to automatically DROP all packets to and from the server.

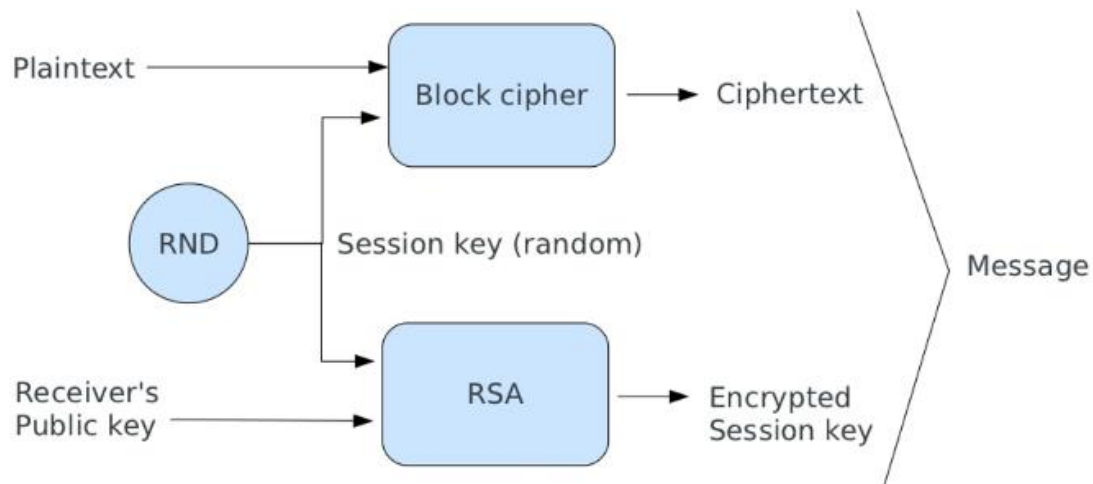


Figure 11 – Hybrid Encryption procedure. [17]

Digital Signature

Digital Signatures are used to validate the authenticity, integrity and non-repudiation of a message. The disadvantage of an asymmetric cryptosystem is that the server's public key is freely distributed and can be misused if retrieved by an undesired attacker. It is therefore important to be able to validate the client that sent the message and ensure the message hasn't been modified or fabricated in some way in transit and ensure the document received is exactly as it was, when the sender signed it. Digital Signature solves the below 3 problems –

- Authentication: a digital signature gives the receiver reason to believe the message was created and sent by the claimed sender.
- Non-repudiation: with a digital signature, the sender cannot deny having sent the message later.
- Integrity: a digital signature ensures that the message was not altered in transit.

Digital Signing is based on asymmetric cryptography. The sender's (plaintext) message is hashed using some hash algorithm. The output is then encrypted using the sender's private key producing the digital signature. The message and signed message are then sent to the desired party.

The receiver can then verify the signature by obtaining the sender's public key (which is freely available). The receiver takes the message and decrypts the digital signature using the sender's public key returning the hash output. If the digital signature cannot be decrypted, then the receiver knows that the digital signature did not come from the desired sender because only the desired sender's public key is able to decrypt the hash generated with sender's private key. When the receiver gets the above hash, they will check the integrity of the message. Receiver hashes the message with the same hash algorithm the sender used and compares the computed hash to the hash received in the message. If the hashes match the receiver can be confident that the message has not changed since the sender signed the message. If the hashes are not equal, then the receiver will know that the message has been changed in transit.

Note that digital signature does not encrypt the message itself. To encrypt the message the above asymmetric cryptography would have to be applied by using the receiver's public key to encrypt.

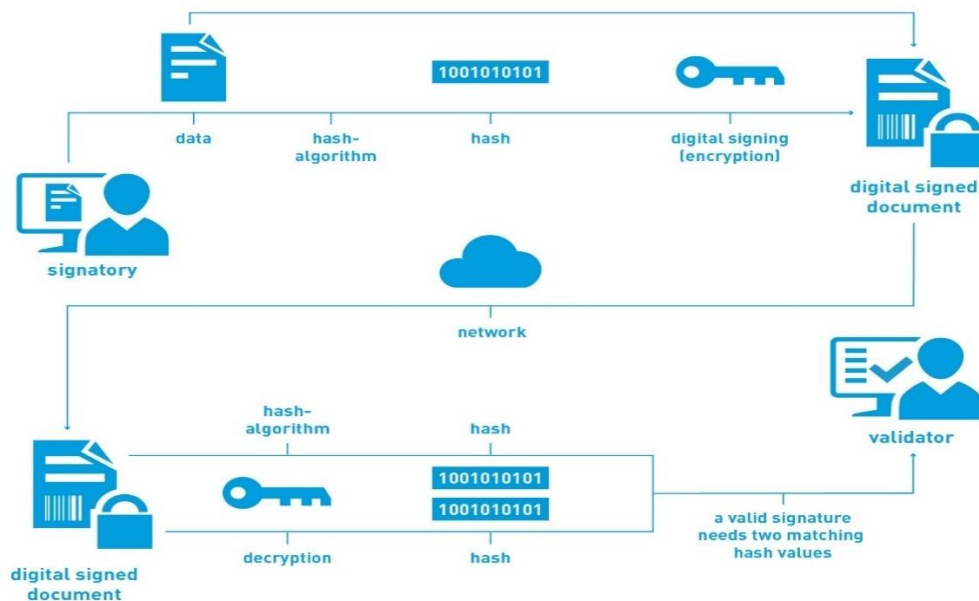


Figure 12 – Digital Signature procedure. [18]

Encrypt-Then-Sign or Sign-Then-Encrypt

When combining our message encryption with digital signature, two options are available. The message can undergo ‘Encrypt-then-Sign’ or ‘Sign-then-Encrypt’.

Both solutions have their slight flaws which will be addressed later. The most appropriate method for our Port Knocking protocol is the ‘Encrypt-then-Sign’ method. This decision is solely based on making our Port Knocking daemon as robust as possible. The ‘Sign-then-Encrypt’ method, if implemented, the daemon will have to first decrypt the message before verifying the digital signature. Port Knocking protocols and servers are vulnerable to DoS and other undesired attacks that will cause server downtime if not correctly handled. If a large volume of invalid packets are sent by an attacker to the server, computing the decryption first before verifying a hash to find out that a packet sent is not valid will only slow down the daemon’s operations and prevent legitimate traffic from reaching the server.

In terms of designing a security protocol that is exposed to such attacks, the ‘Encrypt-then-Sign’ seems more appropriate to meet the needs of the protocol. Computing a hash is much faster than asymmetric decryption. Hence, encrypting the packet first, then digitally signing client-side will make the daemon compute the hash for signature verification first on the incoming Port Knocking packet. If the computed hash does not compare to the senders hash, indicating tampering of a UPD packet through modification or some other malicious attack, the daemon can discard of the packet and the decryption process does not have to be performed which would take up much of the daemons processor and computational time. Therefore, the daemon should be robust and well-prepared to deal with DoS attacks and other attacks that involve high volume traffic that intend to cause server downtime.

Both above approaches have slight downfalls which will be patched in the proposed Port Knocking protocol. Fixing the ‘Encrypt-then-Sign’ method will be focused on as it is the proposed solution for the protocol. One problem occurs when using RSA or El Gamal encryption algorithms as discussed in a sequel to Abadi’s ‘Robustness Principles’ paper, Anderson showed the above ‘Encrypt-Then-Sign’ is dramatically weaker than had been thought as the sender can be left vulnerable to substituted-ciphertext attacks [19].

Thus, whenever we want to sign a ciphertext, Anderson's attack forces the sender to sign, along with the ciphertext, either the plaintext itself or the servers public key.

$$C \rightarrow S : \{\{msg\}^{S_{PUB}}, \#msg\}^{C_{PRIV}}$$

Signing the plaintext alongside the ciphertext gives non-repudiation which allows the receiver assurance that the sender cannot deny the validity of sending a message. Signing the encryption key can be more easily understood as a defence against Anderson's attack. Unfortunately, the above method only makes the illegible ciphertext and not the plaintext non-reputable.

It is relatively easy for any eavesdropper to replace the sender's signature with their own, to claim authorship for the encrypted plaintext. The eavesdropper must block the sender's original message, before sending the re-signed ciphertext. To defend against the above signature modification attack, sign and encrypt must cross-refer. The problem occurs where messaging standards have incorrectly treated public-key encryption and digital signature as if they were fully independent operations. This independence separation is convenient for writing standards and software writing but is cryptographically incorrect. Applying independent operations on top of each other causes the most outer crypt layer to be replaced unbeknownst to both sender and receiver making the security weak. In order to work properly together, the signature layer and the encryption layer must refer to one another. The receiver needs proof that the signer and the encryptor were the same person. Once such a cross-reference is in place, an attacker cannot remove and replace the outermost layer, because the inner layer's reference will reveal the alteration.

The solution to this problem is to prepend the sender's identification onto the message. The message will then be encrypted, and the signature computed. This links the outer layer's key-pair to the inner layer, and prevents an attacker replacing the sender's signature. Encrypting the sender's name proves that the sender performed the encryption: the enclosed name shows that the encryptor intends for the outer signature to carry the same name (the sender's). The outer signature, in turn, says that the sender did indeed touch the ciphertext. Therefore, the receiver knows that the sender performed the encryption.

By prepending the sender's identification, it can be verified that if the signers IP address doesn't match the sender's identification inside the plaintext, the sender can conclude that the message was tampered with. This repair solves 'Encrypt-Then-Sign' non-repudiation problem, since the sender signs the plaintext explicitly. Finally, the repair blocks Andersons plaintext replacing attack. Enclosing both the sender and receiver's identification in the message-body ensures a safer protocol.

$$C \rightarrow S : \{\{C \rightarrow S', msg\}^{S_{PUB}}, \#msg\}^{C_{PRIV}}$$

Fortunately, Hybrid Encryption inherently lends itself to the $\{\{C \rightarrow S', msg\}^{S_{PUB}}, \#msg\}$ procedure. Hashing the message using the generated AES key completes the $\{\#msg\}$ process. RSA encryption of the server's public key is performed on the AES key so to secure the symmetric key forming $\{AES\}^{S_{PUB}}$. However, as shown above, the AES key is used to retrieve the original message allowing, trivially, to say the $\{AES\}^{S_{PUB}}$ process is the above $\{C \rightarrow S', msg\}^{S_{PUB}}$ procedure. Applying digital signature of the client private key to the hybrid encryption the following is achieved for the proposed protocol –

$$C \rightarrow S : \{\{AES\}^{S_{PUB}}, \#msg\}^{C_{PRIV}} \text{ where } msg = \{C \rightarrow S', NTP \text{ timestamp, connection knock } (C_i)\}$$

To avoid confusion the client and server prepended identification will be included in the message for the proposed protocol. This should not increase AES hashing time by a significant amount as the identifications are relatively few bits.

Java Encryption Implementation

For each index in the Port Knocking sequence a packet is sent to the desired server as a single knock. For each single knock a new 128-bit AES symmetric key is generated. The AES key is used to encrypt the data packet string (consisting of NTP timestamp, random connection port, server and client IP address). The AES key is then encrypted using the servers 2048-bit RSA public key. Both the encrypted data string and encrypted key are hashed then digitally signed using the 'SHA256withRSA' algorithm and the clients RSA private key.

A UDP packet is then formed, using the Java 'DatagramPacket' library, consisting of the server IP address, Port Knocking sequence index value as the server port number and the packet data consisting of the encrypted data and the encrypted data signed. The UDP packet is sent out using the 'DatagramSocket' library. [20]

```
// get list of random ports of length 'knockingSequence'
// used as connection ports for client-server connection
// 'knockingSequence' is the Port Knocking sequence inputted by client
ArrayList<Integer> connectionPorts =
    getRandomConnectionSockets(knockSequence.size());
// for each element in PK sequence send an attempt knock packet to server
for (int i = 0; i < knockSequence.size(); i++) {
    // get timestamp from NTP server
    long time = System.currentTimeMillis();

    // message to be encrypted
    // incl. NPT timestamp & random connection port
    // prepend client & server address (authentication purpose)
    String plainText = time + "," + connectionPorts.get(i) + "," +
        this.address.getHostAddress() + "," +
        InetAddress.getLocalHost().getHostAddress();

    // generate 128-bit symmetric AES key and AES encrypt message
    KeyGenerator generator = KeyGenerator.getInstance("AES");
    generator.init(128); // The AES key size in number of bits
    SecretKey secKey = generator.generateKey();

    // encrypt AES key using 2048-bit RSA server public key
    String encryptedKey = Base64.getEncoder().encodeToString(
        RSAEncrypt.encryptKey(publicKey, secKey));

    // AES encrypt the plaintext string
    Cipher aesCipher = Cipher.getInstance("AES");
    aesCipher.init(Cipher.ENCRYPT_MODE, secKey);
    String encryptedString = Base64.getEncoder().encodeToString(
        aesCipher.doFinal(plainText.getBytes()));

    // concatenate encrypted (AES) string & encrypted (RSA) AES key
    String packetData = encryptedString + " " + encryptedKey;
    // digitally sign packetData
    // using 'SHA256withRSA' hash algorithm & client private key
    String signedPacket = digitalSignature(packetData, privateKey);
    // concatenate data packed and signed data packet
    String sendPacket = signedPacket + ";" + packetData;

    // send packet
    buf = sendPacket.getBytes();
    DatagramPacket packet = new DatagramPacket(buf,
        buf.length, address, knockSequence.get(i));
    socket.send(packet);
}
```



```
}
```

The Java RSA encryption method is given below.

```
// RSA encryption method
// Base64 encoded public RSA key to encrypt the (AES) SecretKey
public static byte[] encryptKey(String publicKey, SecretKey secKey)
{
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.ENCRYPT_MODE, getPublicKey(publicKey));
    return cipher.doFinal(secKey.getEncoded());
}
```

The Java Digital Signature method which takes an encrypted string, performs SHA256 hash algorithm on the string and RSA encrypts using the sender's private key.

```
private String digitalSignature(String text, String privKey) {
    // sign encrypted data using clients private key
    Signature sign = Signature.getInstance("SHA256withRSA");
    sign.initSign(RSAEncrypt.getPrivateKey(privateKey));
    sign.update(text.getBytes(UTF_8));
    byte[] signature = sign.sign();
    return Base64.getEncoder().encodeToString(signature);
}
```

Final Decryption Implementation

The Port Knocking daemon holds a directory of client RSA public keys to verify their digital signature appended to the UDP packet of each single knock. A text file or perhaps a more secure file system or database could be used to store all client public keys and their associated IP addresses. For testing purposes, the Port Knocking protocol implemented uses a HashMap to hold client IP addresses as HashMap key and associated string of the client public RSA key as the HashMap value. Therefore, each client IP address can only have one associated public key.

```
private HashMap<InetAddress, String> keyManagment = new HashMap<>();
this.keyManagment.put(InetAddress.getByName("192.168.56.1"), clientPubKey1)
```

The UDP packet data is read in by the buffered reader. The line is parsed out into 2 strings – the packet data and the packet data with digital signature. The digital signature is verified by computing the SHA256 hash on the packet data and RSA decryption is performed on the digital signature using the sender's public key forming a hash value. The resulting hashes are then compared. If the hashes match and signature is verified, the AES key is decrypted from the packet data using the server RSA private key and finally, the encrypted packet string is decrypted using the AES symmetric key. Now that the plaintext is available, the daemon verifies the client and server IP addresses prepended onto the packet to make sure the signer and encryptor were the same person.

```
// read next buffered reader line which holds packet data & split
String[] signatureSplit = inputStreamReader.readLine().split(";");
// byte array of packet data (no signature)
byte[] stringText = signatureSplit[1].toString().getBytes();
// byte array of packet data with signature
byte[] signature = signatureSplit[0].toString().substring(28).getBytes();
// string of packet data and signed packet data
String text = new String(stringText, 0, stringText.length);
String sign = new String(signature, 0, signature.length);
```

```

// server contains clients key but needs to verify signature
if(verify(text, sign, this.keyManagment.get(InetAddress.getByName(srcIP))))
{
    // verify client signature
    // decrypt aesKey using server RSA private key
    // then, decrypt packet data using aesKey
    // parse decrypted packet data

    // parse packet data string
    String[] packetData = text.split(" ");

    // decrypt AES Key
    String aesKey = new String(packetData[1].toString().getBytes(),
        0, packetData[1].toString().getBytes().length);
    byte[] decryptKey = RSAEncrypt.decrypt(serverPrivKey, aesKey);
    SecretKey origionalKey = new SecretKeySpec(decryptKey,
        0, decryptKey.length, "AES");

    // decrypt packet data used to form 'SingleKnock' instance
    Cipher aesCipher = Cipher.getInstance("AES");
    aesCipher.init(Cipher.DECRYPT_MODE, origionalKey);
    byte[] bytePlainText = aesCipher.doFinal(
        Base64.getDecoder().decode(packetData[0].toString()));

    String plainText = new String(bytePlainText);
    String values[] = plainText.split(",");

    // verifies client and server IP addresses prepended to packet data
    // proof that signer and encryptor where the same person
    if(!values[2].equals(InetAddress.getLocalHost().getHostAddress())
        || !values[3].equals(srcIP)) {
        // boolean value that will later disallow connection
        parsingCheck = false;
    }
}
}

```

Method for Digital Signature verification.

```

// method to authenticate the client's digital signature
// compute 'SHA256withRSA' hash on packet data using client RSA private key
// compares output hash on packet data with signed packet data
public boolean verify(String text, String signature, String pubKey) throws
NoSuchAlgorithmException, InvalidKeyException, SignatureException {
    Signature sign = Signature.getInstance("SHA256withRSA");
    sign.initVerify(RSAEncrypt.getPublicKey(pubKey));
    sign.update(text.getBytes(UTF_8));
    byte[] signatureBytes = Base64.getDecoder().decode(signature);
    return sign.verify(signatureBytes);
}

```

Method to RSA decrypt the symmetric AES key.

```

// RSA decryption methods
// Base64 encoded private RSA key to decrypt the (AES) key string
public static byte[] decrypt(String privateKey, String aesKey) {
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.DECRYPT_MODE, getPrivateKey(privateKey));
    return
        cipher.doFinal(Base64.getDecoder().decode(aesKey.getBytes()));
}

```

Network Time Protocol Clock Synchronization

UDP packets tend to deliver out of order. This problem is addressed by NTP (Network Timestamp Protocol) protocol of synchronising both client and server clocks to an NTP server. An alternative approach is to append a sequence number onto the UDP data field and increment the sequence number when the next packet sequence is sent by the client. The server can then reorder packets based on the sequence number if they happen to arrive out-of-order. This solution, although solving the out-of-order delivery problem, it not as multi-purpose as the NTP timestamp solution that the protocol will implement.

One implementation of using Port Knocking with NTP timestamp only gave accuracy to the minute [3]. Such an accuracy would prove useless in distinguishing out of order packet delivery. To make such a protocol secure the timestamp would need to be accurate to the nearest millisecond. This would ultimately fix the problem of out of order packets. If the Port Knocking daemon at the server identifies packet being sent with a timestamp older than a previously received packet from the same client, the server will reorder the packets based on the timestamp included in the packets. Having accuracy of an extended period would allow attackers to easily replay packets or give them a far greater amount of time to perform packet modification to manipulate the Port Knocking protocol

If by chance the timestamp included is a future time, the packet will be silently dropped by the daemon. The reason for a future timestamp may be due to the client's clock not being correctly synced with the NTP server or the client and server are using a different NTP servers which would return a different timestamp.

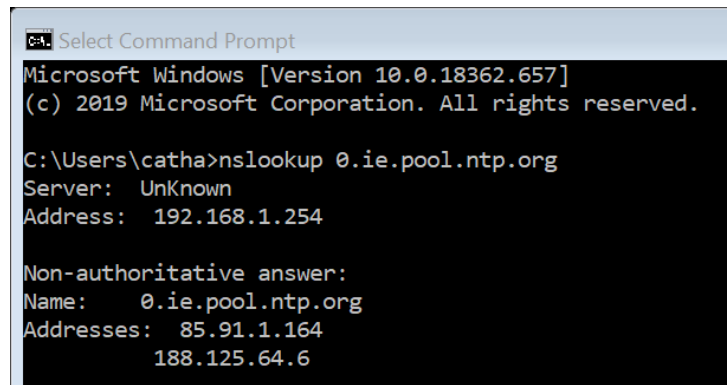
Unfortunately, one problem NTP time synchronization cannot solve is the UDP packet loss problem. A UDP connection is known for not being reliable and does not even guarantee packet delivery to the server. To combat this problem a time interval can exist client side where it will resend the packet sequence again if a connection is unsuccessful. The same time interval will be set for the daemon. If the time interval has been exceeded since the last received packet from the same client, the daemon will assume packet loss has occurred in the network, the buffer for holding the port knocks of that client will be cleared and a new buffer of the port knocks will begin.

Server (Ubuntu 16.04) NTP Clock Synchronization

Connecting to an NTP server through the university network proved to be the biggest difficulty as the university firewall has blocked any NTP connection to external NTP servers. One solution is to use my own mobile hotspot to get around the firewall issues and connect to the external NTP servers available to the public. The problem with this approach is that firewall rules of the server need to be set prior to starting the Port Knocking Daemon. With using the public NTP servers available to Ireland, one NTP server can be assigned multiple publicly static IP addresses.

```
server 0.ie.pool.ntp.org
server 1.ie.pool.ntp.org
server 2.ie.pool.ntp.org
server 3.ie.pool.ntp.org
```

Figure 13 – NTP servers that are publicly available for Ireland. [21]



```

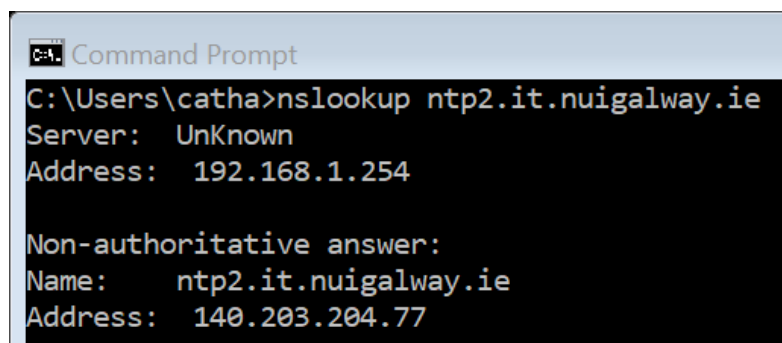
C:\Users\catha>nslookup 0.ie.pool.ntp.org
Server: UnKnown
Address: 192.168.1.254

Non-authoritative answer:
Name: 0.ie.pool.ntp.org
Addresses: 85.91.1.164
           188.125.64.6

```

Figure 14 – Sample screenshot of multiple static public IP that exist for one Ireland NTP server.

Therefore DNS (Domain Name Server) lookup can return a timestamp from any one address assigned to the NTP server. It is highly likely that an IP address assigned will change multiple times when polling an NTP server. Instead of setting multiple firewall rules for each IP address a second approach can be used. NUI, Galway has an NTP server setup from within the university which can be accessed from the DNS name 'ntp2.it.nuigalway.ie' returning one public static IP address '140.203.204.77'.



```

C:\Users\catha>nslookup ntp2.it.nuigalway.ie
Server: UnKnown
Address: 192.168.1.254

Non-authoritative answer:
Name: ntp2.it.nuigalway.ie
Address: 140.203.204.77

```

Figure 15 – NTP server for National University of Ireland, Galway.

Now that a valid NTP server has been retrieved, we can configure our Linux Real-Time Clock (RTC) to use clock synchronization from the above NTP server. Firstly, install the NTP package and amend the NTP config file to set the university NTP server as 'server 140.203.204.77'. Alternatively, the NTP servers available in Ireland can be set as 'server x.ie.pool.ntp.org iburst'. However, each IP address of the server will require both INPUT and OUTPUT firewall rules to be set. It can be said that appending too many rules allowing all NTP synchronization traffic into the server can make the server more vulnerable to attacks. Although, it might be useful to add another NTP server for redundancy purposes in case the above NTP server experiences downtime.

```
# install ntp and edit ntp configuration file
sudo apt-get install ntp -y
sudo nano /etc/ntp.conf
```

```

GNU nano 2.5.3           File: /etc/ntp.conf

# /etc/ntp.conf, configuration for ntpd; see ntp.conf(5) for help
driftfile /var/lib/ntp/ntp.drift

# Enable this if you want statistics to be logged.
#statsdir /var/log/ntpstats/

statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable

# Specify one or more NTP servers.

# Use servers from the NTP Pool Project. Approved by Ubuntu Technical Board
# on 2011-02-08 (LP: #104525). See http://www.pool.ntp.org/join.html for
# more information.
#server 0.ie.pool.ntp.org iburst
#server 1.ie.pool.ntp.org iburst
#server 2.ie.pool.ntp.org iburst
#server 3.ie.pool.ntp.org iburst

server 140.203.204.77

```

Figure 16 – The ‘ntp.conf’ file that contains and loads the NTP server for application use.

The firewall rules for the NTP server are required to be set to allow INPUT and OUTPUT traffic to and from the NTP server. Below are the default iptables rules to be set on server start-up allowing –

- DROP all INPUT, OUTPUT and FORWARD traffic.
- OUTPUT UDP traffic from Port Knocking server to NTP server with destination port 123.
- INPUT UDP traffic from NTP server to Port Knocking server with source port 123.
- OUTPUT and INPUT localhost UDP traffic accepted.
- Save the above firewall rules.

```

# set default iptables rules to DROP
sudo iptables -P INPUT DROP
sudo iptables -P OUTPUT DROP
sudo iptables -P FORWARD DROP
# ACCEPT all NTP and localhost INPUT and OUTPUT traffic
sudo iptables -A INPUT -s <ntp_server_ip_addr> -p udp -m udp --sport 123 -j ACCEPT
sudo iptables -A INPUT -s 127.0.0.1 -d 127.0.0.1 -p udp -j ACCEPT
sudo iptables -A OUTPUT -d <ntp_server_ip_addr> -p udp -m udp --dport 123 -j ACCEPT
sudo iptables -A OUTPUT -s 127.0.0.1 -d 127.0.0.1 -p udp -j ACCEPT
# install package and save iptables rules for OS boot
sudo apt-get install iptables-persistent -y
sudo netfilter-persistent save
sudo netfilter-persistent reload

```

```

cc3@cc3-VirtualBox: ~
GNU nano 2.5.3      File: /etc/iptables/rules.v4

# Generated by iptables-save v1.6.0 on Mon Mar  2 19:07:51 2020
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [4:292]
-A INPUT -s 140.203.204.77/32 -p udp -m udp --sport 123 -j ACCEPT
-A INPUT -s 127.0.0.1/32 -d 127.0.0.1/32 -p udp -j ACCEPT
-A OUTPUT -d 140.203.204.77/32 -p udp -m udp --dport 123 -j ACCEPT
-A OUTPUT -s 127.0.0.1/32 -d 127.0.0.1/32 -p udp -j ACCEPT
COMMIT
# Completed on Mon Mar  2 19:07:51 2020

```

Figure 17 – Shows iptables config file to load firewall commands on VM start-up for NTP clock synchronization.

```

cc3@cc3-VirtualBox: ~$ sudo iptables -L
[sudo] password for cc3:
Chain INPUT (policy DROP)
target      prot opt source                destination           udp spt:ntp
ACCEPT      udp  --  140.203.204.77         anywhere              udp spt:ntp
ACCEPT      udp  --  localhost             localhost

Chain FORWARD (policy DROP)
target      prot opt source                destination

Chain OUTPUT (policy DROP)
target      prot opt source                destination           udp dpt:ntp
ACCEPT      udp  --  anywhere              140.203.204.77        udp dpt:ntp
ACCEPT      udp  --  localhost             localhost
cc3@cc3-VirtualBox:~$

```

Figure 18 – Firewall iptables rules loaded for NTP clock synchronization.

Now that the firewall rules have been set and the NTP server configured, NTP clock synchronization can now be started. These steps only needed to be done once and will be set every time the Ubuntu machine is booted.

```

cc3@cc3-VirtualBox: ~$ sudo service ntp restart
cc3@cc3-VirtualBox:~$ sudo service ntp status
● ntp.service - LSB: Start NTP daemon
   Loaded: loaded (/etc/init.d/ntp; bad; vendor preset: enabled)
   Active: active (running) since Tue 2020-03-10 11:45:18 GMT; 11s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 2620 ExecStop=/etc/init.d/ntp stop (code=exited, status=0/SUCCESS)
  Process: 2632 ExecStart=/etc/init.d/ntp start (code=exited, status=0/SUCCESS)
    CGroup: /system.slice/ntp.service
            └─2644 /usr/sbin/ntpd -p /var/run/ntpd.pid -g -u 122:131

Mar 10 11:45:18 cc3-VirtualBox systemd[1]: Stopped LSB: Start NTP daemon.
Mar 10 11:45:18 cc3-VirtualBox systemd[1]: Starting LSB: Start NTP daemon...
Mar 10 11:45:18 cc3-VirtualBox ntp[2632]: * Starting NTP server ntpd
Mar 10 11:45:18 cc3-VirtualBox ntpd[2641]: ntpd 4.2.8p4@1.3265-o Tue Jan  7 15:0
Mar 10 11:45:18 cc3-VirtualBox ntpd[2641]: Command line: /usr/sbin/ntpd -p /var/
Mar 10 11:45:18 cc3-VirtualBox ntp[2632]: ...done.
Mar 10 11:45:18 cc3-VirtualBox systemd[1]: Started LSB: Start NTP daemon.
Mar 10 11:45:18 cc3-VirtualBox ntpd[2644]: proto: precision = 0.108 usec (-23)
Mar 10 11:45:18 cc3-VirtualBox ntpd[2644]: switching logging to file /var/log/nt
lines 1-18/18 (END)

```

Figure 19 – Start NTP service and show active status.

If NTP synchronization has successfully occurred the following similar outputs will be shown.

```

cc3@cc3-VirtualBox: ~
cc3@cc3-VirtualBox:~$ ntpq -pn
remote          refid      st t when poll reach  delay  offset  jitter
=====
140.203.204.77 .GPS.      1 u  53   64    1  41.499 -1444.8   0.000
cc3@cc3-VirtualBox:~$

```

Figure 20 – The 'ntpq' output showing running NTP servers.

It is observed the above polling of the NTP server is every 64 milliseconds. The output shows the NTP server being used is a Global Positioning System (GPS) NTP server. Using the NTP servers available to Ireland are Public NTP servers. A GPS NTP server can provide the following benefits [22] –

- GPS broadcasts are direct and continuous. Allows a client to keep sampling data and maintain a precise clock calibration. Public NTP servers only allow polling every 4 seconds.
- GPS time is orders of magnitude more accurate than internet alternatives and timing received exhibits less variation. With web-based time servers, a signal may work through poor internet connections and latency limiting the precision on the internet-based time.

Note that once an operating system has been set up and synchronised with an NTP server, the operating system's clock will now use the NTP server time instead of the Real Time Clock (RTC). The Java 'System.currentTimeMillis()' command will now retrieve the NTP clock time.

```

cc3@cc3-VirtualBox: ~
cc3@cc3-VirtualBox:~$ timedatectl
Local time: Mon 2020-03-16 14:32:56 GMT
Universal time: Mon 2020-03-16 14:32:56 UTC
RTC time: Mon 2020-03-16 14:32:57
Time zone: Europe/Dublin (GMT, +0000)
Network time on: no
NTP synchronized: yes
RTC in local TZ: no
cc3@cc3-VirtualBox:~$

```

Figure 21 – The 'timedatectl' showing that the RTC clock is running NTP clock synchronization.

Client (Windows 10) NTP Clock Synchronization

Under the 'Registry Editor' program go to the following directory where all timer servers are contained –

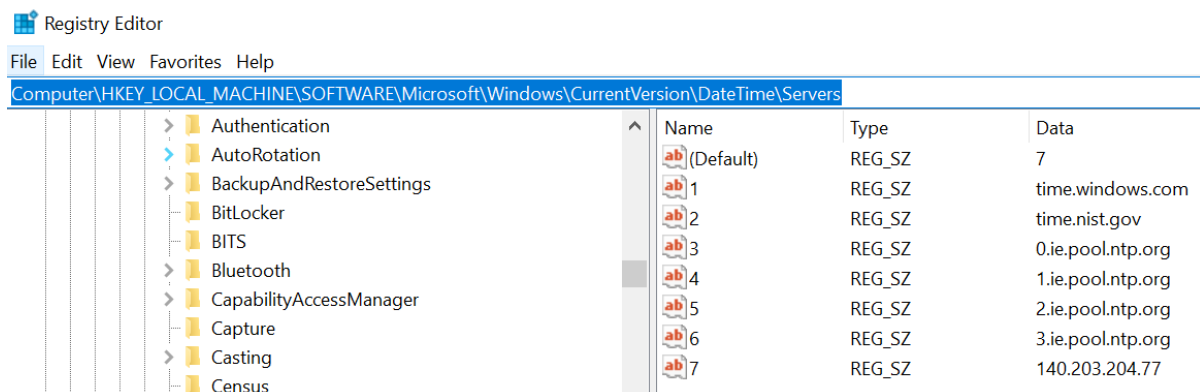


Figure 22 – Setting up the host machine (Windows 10) with an NTP server.

Our NTP server can be added to the list with the assigned IP address '140.203.204.77'. Now we have added the NTP server, the Windows system clock needs to be configured to use that NTP clock. Therefore, go to 'Date & Time Settings' and into 'Add clocks for different time zones'. Under

'Internet Time' tab go to 'Change Settings'. Tick the box 'Synchronize with an Internet time server' and select your desired NTP server just added '140.203.204.77' from the dropdown menu and 'Update now'. Now the Windows 10 environment has been configured to use the desired NTP time server to synchronise its clocks.

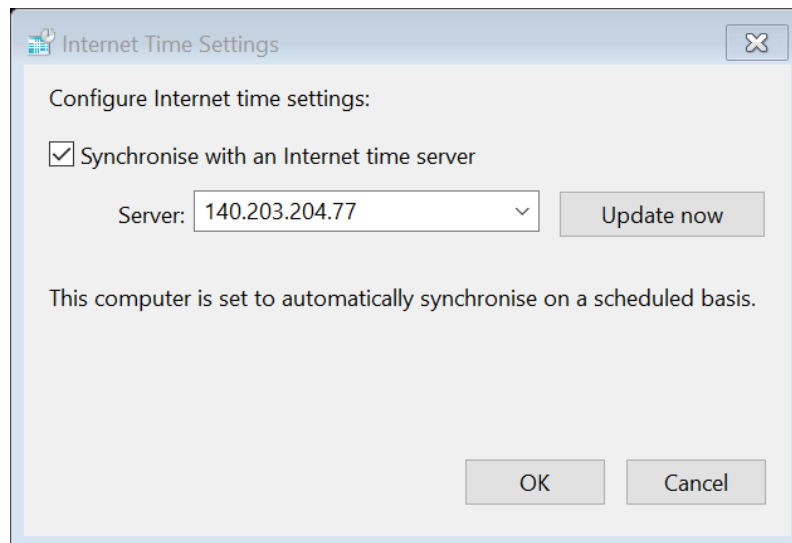


Figure 23 – Setting RTC clock of host machine (Windows 10) to using NTP clock synchronization.

Packet Capture

To overcome the server-side packet capture problem one solution is to use the 'Packet Capture for Java' (Pcap4J) or 'Java Packet Capture' (Jpcap [23]) opensource libraries allowing Java programmers to capture network packets on their specified network interface. The 'Jpcap' library has not been maintained for some time and known bugs exist in the library due to vulnerabilities not being patched. The relatively new 'Pcap4J' library is maintained on a regular basis and the library can run on both Linux and Windows OS which seemed an obvious choice for my application. However, it can often happen that these libraries become less maintained overtime causing out-of-date versions to become vulnerable to attacks. Having to design a security protocol, this is not something which we would like to happen. Additionally, both libraries have huge dependencies on build tools maven/gradle and libpcap/WinPcap libraries. Also, to get a basic version of 'Pcap4J' [24] working on the Windows 10 environment multiple Dynamic-Link Library (DLL) files needed to be copied into Windows executable directory. Issues also arose installing jar files and setting build and class paths.

The reliable TcpDump library presented itself to be the best alternative. TcpDump was originally only available on Linux system can now be installed on Windows environments via Command Prompt. Because majority of servers nowadays are Linux-based, TcpDump is an appropriate tool to use for our protocol. TcpDump is usually already preinstalled on the OS and relatively easy to use. The below code shows how the TcpDump command is constructed and run by the Java Port Knocking daemon. The command is configured to capture UDP packets on the inputted Port Knocking sequence without TcpDump buffering and print ASCII packet data to parse source, destination and data information. TcpDump default buffers on packet capturing and will only print an output until the buffer is full.

Note that TcpDump output contains 28 characters of padding added by TcpDump printout. The Java program will need to parse out this padding to get the real packet data. Failure to do so will cause an error when decrypting. The added padding is shown in red below.


```

cc3@cc3-VirtualBox:~/git/Port-Knocking-Protocol$ sudo tcpdump -A -l -n udp port '(5 or 7000 or 4000 or 6543)'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
15:40:58.652071 IP 192.168.56.1.54610 > 192.168.56.101.5: UDP, length 389
E.....8...8e.R.....iAheA1iE9hmTwUaFuOhA75AgtoVtviFZhrR3Jx58YVrM= IH5YVDJy+6hmYRk0J/qBiXhv8xHnXLRJGR+8oVmlcGuyq6LxJuPAWbiI/Y6
rf95mNBsRCxFFJQdzoniHlqsgRm0156Wk+PbNMUQE1zhtOG9Lk+ZksVs60xPuwRTElno8RiCutcitMJz0BG5fjDgn0Jsy9RrqEzDTNi8EZVroMa0n2yq9RCouqpkF2f0j72nD
0C4LLeA==
15:40:58.654062 IP 192.168.56.1.54610 > 192.168.56.101.7000: rx type 89 (389)
E.....8...8e.R.X..Q.5YeVq1Q489uwvoaizYKAY72KXv25GBoynCR26YtZeQ0= N5FBTVgSFbM/CHqD3W/UKFwKZB8c/AOZI0ufB60pboJ1pkUA8JfwuJZvLVJF
+ZkbK0R7ytUESwRC6LHPKzWFXbnfhaFx/kZ0oc2TdLhtQ8WDL51BFzYDCCN/AKDuo4ax3un2nrgDMQF5Bk3RnWVbEL/xSKFKNXFLQaxL06Icqb9LC7+xPFFhTFm9FGoh+KYM
/RIUDyA==
15:40:58.655322 IP 192.168.56.1.54610 > 192.168.56.101.4000: UDP, length 389
E.....8...8e.R.....tfjMTAcJqgVgicyFU/DqWtKMVbdH1aVYFwvRR3VMsto= g8QvoqseEYeLV2sA7oCJTyxkcvDbaTp35NulgnC9BtKKKWFVesNavcrH3sz
wuwpFdu+C2nYueK35PZY0N6C9L+x1ZegPX8trWOPMUC/NZtmkIdsuWi/r5TM3r56z0+BasTMvnqzJQcglRQSaKmh9Sgh0KywxUdw1H4QW0HvKpmqThDW1H5HC+hnmLI2H53n
eN3nDHg==
15:40:58.658841 IP 192.168.56.1.54610 > 192.168.56.101.6543: UDP, length 389
E.....8...8e.R.....azkRTTj7dE+sucTP9wY6+VI+mC4jd5/4QPlcY2quGLY= a6Z7PwvIbsYjwhXlIE2/N7xKzYEZgyvbcXANbiYkMumzaceBnHYh5DrD72CE
9jxv8THUxROIyd2ZHuHJXv1SuLaNcD5j6D4jNVHcg7x0T5+nXDS+06ruZqT+sCNTQ/1Xc7cstYmQNXEAW/p4oYVA0Ta06cLeihxhCc4CNm0yJlJlSNca80c7LVpqTAXwOUKCD
wMfKZ7W==

```

Figure 24 – Sample TcpDump output of UDP packets for specific server ports.

TcpDump Flag	Description [25]
-A	Print each packet (without link level header) in ASCII.
-l	Flushes TcpDump buffer on message arrival.
-n	Don't convert IP addresses to DNS name.

Table 6 – TcpDump flag descriptions.

Port Knocking daemon constructor forming the TcpDump command based on the Port Knocking sequence inputted by the user.

```

// port knocking server constructor
public MyServer(int knockPort, ArrayList<Integer> knockSequence) {
    // corresponding port to port knock sequence
    this.confirmKnockingPort = knockPort;
    this.confirmKnockingSequence = knockSequence;
    // tcpdump command constructed from inputted port knock sequence
    String command = "/usr/sbin/tcpdump -A -l -n udp port '(";
    for(int i = 0; i < confirmKnockingSequence.size(); i++) {
        if(i == confirmKnockingSequence.size() -1) {
            command += confirmKnockingSequence.get(i) + ")";
        } else {
            command += confirmKnockingSequence.get(i) + " or ";
        }
    }
    this.tcpDumpCommand = command;
}

```

Running of the TcpDump packet capture command using the Java library 'ProcessBuilder', which is a class used to create operating system processes and can run commands through the 'ProcessBuilder.start()' command as seen below. Another alternative to run the TcpDump command was to use the Java 'Runtime' library. However, it seems 'ProcessBuilder' is preferred and more reliable standard.

```

// setup Java process builder to running tcpdump command on Linux
ProcessBuilder tcpDumpProcessBuilder =
    new ProcessBuilder("/bin/bash", "-c", tcpDumpCommand);
tcpDumpProcessBuilder.redirectErrorStream(true);

//start tcpdump command
Process process = tcpDumpProcessBuilder.start();

```

Server side, the packets are configured in the firewall to be dropped. However, the daemon monitors the network interface for traffic that corresponds to the running TcpDump command. Each Port Knocking sequence from the client causes an 'AttemptKnockingSequence' class instance to be

created. The 'AttemptKnockingSequence' class is identified by the client source port number and client IP address.

```
// AKS constructor identified by client ip address and port number
public AttemptKnockingSequence(InetAddress address, int port) {
    if(address.equals(null) || port == 0) {
        throw new IllegalArgumentException();
    } else {
        this.address = address;
        this.port = port;
    }
}
```

Each Port Knocking sequence from the client causes a 'SingleKnock' class instance to be created.

```
// SingleKnock constructor
// Identified by portKnock, timestamp, connectionKnock and arrivalTime
public SingleKnock(int portKnock, long ntpTime,
    int connectionKnock, long arrivalTime) {

    if(portKnock == 0 || ntpTime == 0 ||
        connectionKnock == 0 || arrivalTime == 0 ) {

        throw new IllegalArgumentException();
    } else {
        this.portKnock = portKnock;
        this.ntpTime = ntpTime;
        this.connectionKnock = connectionKnock;
        this.arrivalTime = arrivalTime;
    }
}
```

The main program contains a HashMap to hold the client 'AttemptKnockingSequence' instance as HashMap key and associated Array List of Port Knocking 'SingleKnocks' as the HashMap value.

```
private HashMap<AttemptKnockingSequence, ArrayList<SingleKnock>> hashKnock
= new HashMap<AttemptKnockingSequence, ArrayList<SingleKnock>>();
```

Parse source and destination IP addresses and port numbers from TcpDump command.

```
// buffered & input stream reader for reading tcpdump command output
BufferedReader inputStreamReader =
new BufferedReader(new InputStreamReader(process.getInputStream()));
// string for reading buffered reader
String line = null;
// read outputs from buffered reader
while ((line = inputStreamReader.readLine()) != null) {
    // packet arrival time
    long arrivalTime = System.currentTimeMillis();
    // parse source and destination information from tcpdump
    String[] tcpArr = line.split(" ");
    // parse source IP address
    int srcIndex = tcpArr[2].toString().lastIndexOf(".");
    String srcIP = tcpArr[2].toString().substring(0,srcIndex);
    // parse source port
    String srcPort = tcpArr[2].toString().substring(srcIndex+1);
    // parse destination IP address
    int destIndex = tcpArr[4].toString().lastIndexOf(".");
```

```
String destIP = tcpArr[4].toString().substring(0, destIndex);
// parse destination port
String destPort = tcpArr[4].toString().substring(
    destIndex+1, tcpArr[4].toString().length()-1);
```

Above values used for the creation of client 'AttemptKnockingSequence' class instance.

```
// set up new AKS with client IP and port if does not already exist
AttemptKnockingSequence aks = null;
try {
    aks = new AttemptKnockingSequence(InetAddress.getByName(srcIP),
Integer.parseInt(srcPort));
} catch (IllegalArgumentException ex) {
    logger.warning("Failure to create client identity (parsing client
IP and/or port): IP - " + aks.getAddress() + ": Port - " + aks.getPort());
}
```

The following overridden methods are called in the 'AttemptKnockingSequence' class when creating a new 'AttemptKnockingSequence' class instance. If an instance has been previously created with the same port and IP address values the 'equals' method will return true and the 'hashCode' method will return the hash of the already created class instance instead of creating duplicate objects.

```
// override equals method
// check equality (by ip and port) of AKS class object instances
@Override
public boolean equals(Object obj) {
    if (obj == null) return false;
    if (!(obj instanceof AttemptKnockingSequence))
        return false;
    if (obj == this)
        return true;
    AttemptKnockingSequence aks = (AttemptKnockingSequence) obj;
    return Objects.equals(address, aks.address) && port == aks.port;
}

// override hashCode
// return hash of AKS object class instance if previously created
@Override
public int hashCode() {
    return Objects.hash(address, port);
}
```

Daemon reads the next line of the TcpDump command which contains the UDP packet data which is used to create a 'SingleKnock' class instance. The line is parsed out into 2 strings – the packed data and the packet data with digital signature. The signature is verified, AES key is decrypted from the packet data using the server RSA private key and finally, the encrypted packet string is decrypted using the AES symmetric key. The encrypted string is then parsed out to hold 'SingleKnock' values in a string array.

Above values are used for the creation of 'SingleKnock' class instance.

```
// add incoming knock to the attempt
SingleKnock single = null;
try {
    single = new SingleKnock(Integer.parseInt(destPort),
Long.parseLong(values[0]), Integer.parseInt(values[1]), arrivalTime);
} catch (IllegalArgumentException ex) {
```

```

        logger.warning("Failure to create single knock (parsing client
packet data): IP - " + aks.getAddress() + ": Port - " + aks.getPort());
    }

```

If the incoming client 'AttemptKnockingSequence' is not contained a HashMap key (i.e. has not had a previous attempt Port Knocking connection with the server), add the client 'AttemptKnockingSequence' and an empty 'SingleKnock' Array List to the HashMap as key-value pair.

```

// check is client AKS does not already
if(!hashKnock.containsKey(aks)) {
    // HashMap value arraylist to hold the SingleKnock class instances
    ArrayList<SingleKnock> knockList = new ArrayList<>();
    // if client AKS does not exist yet -> add to HashMap
    hashKnock.put(aks, knockList);
}

```

Adding of the 'SingleKnock' of the corresponding HashMap key (client 'AttemptKnockingSequence') to the HashMap value Array List.

```

// get HashMap Array List of SingleKnock
ArrayList<SingleKnock> arr = hashKnock.get(aks);
// adding Singleknock to HashMap of corresponding client AKS
hashKnock.computeIfAbsent(aks, k -> new ArrayList<>()).add(single);

```

Client-Server Connection

Server-side, once the daemon has recognised a successful Port Knocking sequence a client-server connection begins. The traditional way to initiate a Port Knocking connection is to open the firewall to the knocking client IP address allowing them access to the corresponding Port Knocking sequence port number which the client desires access to. This type of connection can expose the server to attackers. Through eavesdropping an attacker could potentially discover the Port Knocking sequence if they recognise a pattern of packets trying to access closed ports. Then perhaps they recognise an established client- server connection on a specific port. Unfortunately, the Port Knocking daemon and server have now been compromised allowing an eavesdropper easy access to the server through the Port Knocking sequence and corresponding connection port.

The above scenario is only a possibility of how the protocol could become exposed, but it is important to reduce these vulnerabilities. The proposed protocol intends to allow the Port Knocking sequence of UDP packs on the closed ports as discussed earlier. However, in each UDP packet a random port (between 0 – 65535) is included under the data header. The daemon decrypts each knocking packet to reveal the connection port number. Once a Port Knocking sequence of N packets has been successfully received from a valid client, the daemon should also have N random connection ports from the corresponding client. These ports will be used for the client-server connection allowing the client to connect through one of these connection ports to the corresponding Port Knocking connection port which the client wishes to get access to. Any output traffic intended for the corresponding client will come from the Port Knocking connection port, routed to the available temporary connection port number and sent back to the client as if each packet had come from the latter port. Each N connection ports will get a specific amount of time configured in the firewall, routing traffic to the desired Port Knocking connection port and sending packets back to the client before the next connection packet is configured.

For example, if a Port Knocking sequence of 4 packets existed and each port connection is configured for 15 minutes, the total client-server connection would be one hour. The above solution intends to protect the Port Knocking connection port. Only a valid client who has sent the Port Knocking sequence will connect to the desired service through the random connection ports encrypted in each of the knocking packets. An attacker should never be able identify the corresponding Port Knocking connection port through port scanning or other measures as the connection port will be closed to external traffic and only open to a client that has sent the correct Port Knocking sequence and is accessing the desired port through random ports which the firewall routes to the desired server port.

Below is an example of firewall rules to route traffic from port 3000 to 23 (telnet). Therefore, a client can access telnet from port 3000. Trying to telnet to port 23 will not be successful. [26]

```
# mark incoming packets from client 192.168.56.1 to server port 3000
iptables -t mangle -A PREROUTING -p tcp -s 192.168.56.1 --dport 3000 -j
MARK --set-mark 1

# route packets of client 192.168.56.101 from server port 3000 to 23
iptables -t nat -A PREROUTING -p tcp -s 192.168.56.1 --dport 3000 -j
REDIRECT --to-port 23

# only accept TCP marked packet packets from client 192.168.56.1 to server
port 23 (previously routed from port 3000 to 23)
iptables -A INPUT -p tcp -s 192.168.56.1 --dport 23 -m state --state
NEW,ESTABLISHED -m mark --mark 1 -j ACCEPT

# only send back TCP packets to client from port 23 on port 3000
iptables -A OUTPUT -p tcp --sport 23 -m state --state ESTABLISHED -j ACCEPT
```

Firewall rules configured in Linux Ubuntu 16.04 environment.

```
cc3@cc3-VirtualBox: ~
cc3@cc3-VirtualBox:~$ sudo iptables -L
Chain INPUT (policy DROP)
target    prot opt source                destination
ACCEPT    tcp  --  192.168.56.1            anywhere             tcp dpt:telnet state NEW,ESTABLISHED mark match 0x1

Chain FORWARD (policy DROP)
target    prot opt source                destination

Chain OUTPUT (policy DROP)
target    prot opt source                destination
ACCEPT    tcp  --  anywhere               anywhere             tcp spt:telnet state ESTABLISHED
cc3@cc3-VirtualBox:~$ sudo iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target    prot opt source                destination
REDIRECT  tcp  --  192.168.56.1            anywhere             tcp dpt:3000 redir ports 23

Chain INPUT (policy ACCEPT)
target    prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target    prot opt source                destination

Chain POSTROUTING (policy ACCEPT)
target    prot opt source                destination
cc3@cc3-VirtualBox:~$ sudo iptables -t mangle -L
Chain PREROUTING (policy ACCEPT)
target    prot opt source                destination
MARK      tcp  --  192.168.56.1            anywhere             tcp dpt:3000 MARK set 0x1

Chain INPUT (policy ACCEPT)
target    prot opt source                destination

Chain FORWARD (policy ACCEPT)
target    prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target    prot opt source                destination
```

Figure 25 – Sample iptables firewall rules for client-server connection.

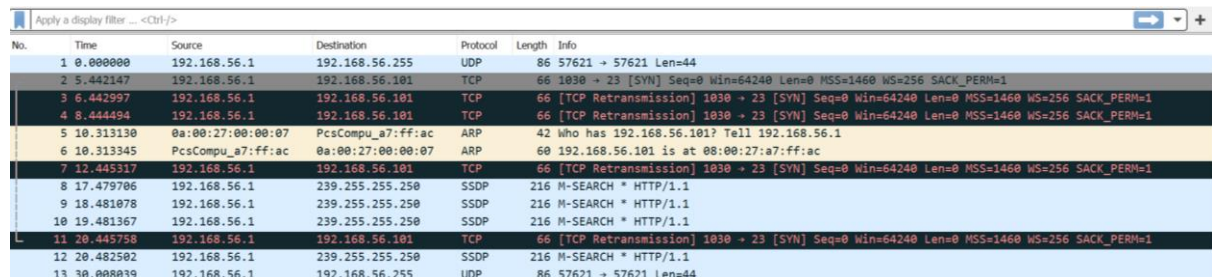
Accessing telnet from port 23 unsuccessful. Wireshark capture of unsuccessful telnet also included.

```
C:\> Command Prompt
Microsoft Windows [Version 10.0.18362.719]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\catha>telnet 192.168.56.101
Connecting To 192.168.56.101...Could not open connection to the host, on port 23: Connect failed

C:\Users\catha>telnet 192.168.56.101 3000
```

Figure 26 – Unsuccessful telnet attempt to Port Knocking server port 23.

A Wireshark packet capture showing a failed telnet connection to port 23. The capture includes several ARP requests and TCP SYN/ACK exchanges. The first TCP SYN packet (No. 2) is from 192.168.56.1 to 192.168.56.101 on port 23. The corresponding ACK packet (No. 3) is received. However, the next SYN packet (No. 4) is a retransmission. The final packet (No. 13) is a UDP packet from 192.168.56.1 to 192.168.56.255.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.56.1	192.168.56.255	UDP	86	57621 → 57621 Len=44
2	5.442147	192.168.56.1	192.168.56.101	TCP	66	1030 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
3	6.442997	192.168.56.1	192.168.56.101	TCP	66	[TCP Retransmission] 1030 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
4	8.444424	192.168.56.1	192.168.56.101	TCP	66	[TCP Retransmission] 1030 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
5	10.313130	0a:00:27:00:00:07	PcsCompu_a7:ff:ac	ARP	42	Who has 192.168.56.101? Tell 192.168.56.1
6	10.313345	PcsCompu_a7:ff:ac	0a:00:27:00:00:07	ARP	60	192.168.56.101 is at 08:00:27:a7:ff:ac
7	12.445317	192.168.56.1	192.168.56.101	TCP	66	[TCP Retransmission] 1030 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
8	17.479706	192.168.56.1	239.255.255.250	SSDP	216	M-SEARCH * HTTP/1.1
9	18.481078	192.168.56.1	239.255.255.250	SSDP	216	M-SEARCH * HTTP/1.1
10	19.481367	192.168.56.1	239.255.255.250	SSDP	216	M-SEARCH * HTTP/1.1
11	20.445758	192.168.56.1	192.168.56.101	TCP	66	[TCP Retransmission] 1030 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
12	20.482502	192.168.56.1	239.255.255.250	SSDP	216	M-SEARCH * HTTP/1.1
13	30.008039	192.168.56.1	192.168.56.255	UDP	86	57621 → 57621 Len=44

Figure 27 – Wireshark capture of unsuccessful telnet attempt.

Accessing telnet from port 3000 successfully. Wireshark capture of successful telnet also included.

```
Telnet 192.168.56.101
Ubuntu 16.04.6 LTS
cc3-VirtualBox login: cc3
Password:
Last login: Mon Jan 27 13:37:36 GMT 2020 from 192.168.56.1 on pts/18
Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.15.0-74-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

0 packages can be updated.
0 updates are security updates.

*** System restart required ***
cc3@cc3-VirtualBox:~$ ifconfig
enp0s3      Link encap:Ethernet  HWaddr 08:00:27:a7:ff:ac
            inet addr:192.168.56.101  Bcast:192.168.56.255  Mask:255.255.255.0
            inet6 addr: fe80::cccd:3dc0:2312:73dd/64  Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:129 errors:0 dropped:0 overruns:0 frame:0
            TX packets:81 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:10441 (10.4 KB)  TX bytes:7645 (7.6 KB)
```

Figure 28 – Successful telnet attempt to Port Knocking server on connection port 3000.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.56.1	239.255.255.250	SSDP	216	M-SEARCH * HTTP/1.1
2	0.434186	192.168.56.1	192.168.56.101	TCP	66	1049 → 3000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
3	0.434400	192.168.56.101	192.168.56.1	TCP	66	3000 → 1049 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
4	0.434464	192.168.56.1	192.168.56.101	TCP	54	1049 → 3000 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
5	0.438907	192.168.56.101	192.168.56.1	TCP	66	3000 → 1049 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=12
6	0.471712	192.168.56.1	192.168.56.101	TCP	60	1049 → 3000 [PSH, ACK] Seq=1 Ack=13 Win=2102272 Len=6
7	0.471905	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [ACK] Seq=13 Ack=7 Win=64256 Len=0
8	0.471937	192.168.56.1	192.168.56.101	TCP	63	1049 → 3000 [PSH, ACK] Seq=7 Ack=13 Win=2102272 Len=9
9	0.472033	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [ACK] Seq=13 Ack=16 Win=64256 Len=0
10	0.472134	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [PSH, ACK] Seq=13 Ack=16 Win=64256 Len=3
11	0.472215	192.168.56.1	192.168.56.101	TCP	63	1049 → 3000 [PSH, ACK] Seq=16 Ack=16 Win=2102272 Len=9
12	0.472326	192.168.56.101	192.168.56.1	TCP	66	3000 → 1049 [PSH, ACK] Seq=16 Ack=25 Win=64256 Len=12
13	0.472406	192.168.56.1	192.168.56.101	TCP	60	1049 → 3000 [PSH, ACK] Seq=25 Ack=28 Win=2102272 Len=6
14	0.516701	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [ACK] Seq=28 Ack=31 Win=64256 Len=0
15	0.516765	192.168.56.1	192.168.56.101	TCP	64	1049 → 3000 [PSH, ACK] Seq=31 Ack=28 Win=2102272 Len=10
16	0.516963	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [ACK] Seq=28 Ack=41 Win=64256 Len=0
17	0.517373	192.168.56.101	192.168.56.1	TCP	66	3000 → 1049 [PSH, ACK] Seq=28 Ack=41 Win=64256 Len=12
18	0.517650	192.168.56.1	192.168.56.101	TCP	57	1049 → 3000 [PSH, ACK] Seq=41 Ack=40 Win=2102272 Len=3
19	0.560613	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [ACK] Seq=40 Ack=44 Win=64256 Len=0
20	0.560645	192.168.56.1	192.168.56.101	TCP	63	1049 → 3000 [PSH, ACK] Seq=44 Ack=40 Win=2102272 Len=9
21	0.560888	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [ACK] Seq=40 Ack=53 Win=64256 Len=0
22	0.561134	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [PSH, ACK] Seq=40 Ack=53 Win=64256 Len=6
23	0.561237	192.168.56.1	192.168.56.101	TCP	57	1049 → 3000 [PSH, ACK] Seq=53 Ack=46 Win=2102272 Len=3
24	0.561469	192.168.56.101	192.168.56.1	TCP	96	3000 → 1049 [PSH, ACK] Seq=46 Ack=56 Win=64256 Len=42
25	0.561518	192.168.56.1	192.168.56.101	TCP	57	1049 → 3000 [PSH, ACK] Seq=56 Ack=88 Win=2102272 Len=3
26	0.604673	192.168.56.101	192.168.56.1	TCP	60	3000 → 1049 [ACK] Seq=88 Ack=59 Win=64256 Len=0

Figure 29 – Wireshark capture of successful connection attempt on connection port 3000.

Port scanning of port 23 (telnet) has a STATE 'filtered' which means 'nmap' cannot determine if the port is open because filtering prevents its probes from reaching the port.

Port scanning of port 3000 has STATE 'open' means a port is actively accepting connections. The scan gives no indication of routing to the telnet service with SERVICE header being 'ppp' (point-to-point protocol).

```

C:\Users\catha>nmap 192.168.56.101 -p 23
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-14 13:03 GMT Standard Time
Nmap scan report for 192.168.56.101
Host is up (0.0010s latency).

PORT      STATE      SERVICE
23/tcp    filtered  telnet
MAC Address: 08:00:27:A7:FF:AC (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 0.86 seconds

C:\Users\catha>nmap 192.168.56.101 -p 3000
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-14 13:04 GMT Standard Time
Nmap scan report for 192.168.56.101
Host is up (0.0010s latency).

PORT      STATE      SERVICE
3000/tcp  open      ppp
MAC Address: 08:00:27:A7:FF:AC (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 0.42 seconds

C:\Users\catha>

```

Figure 30 – Port Scan of telnet port 23 and connection port 3000.

PORT STATE	DESCRIPTION [27]
open	The device is actively accepting TCP connections, UDP datagrams or SCTP associations on this port. Finding these is often the primary goal of port scanning. Security-minded people know that each open port is an avenue for attack. Attackers and pen-testers want to exploit the open ports, while administrators try to close or protect them with firewalls without thwarting legitimate users. Open ports are also interesting for non-security scans because they show services available for use on the network.

filtered	Nmap cannot determine whether the port is open because packet filtering prevents its probes from reaching the port. The filtering could be from a dedicated firewall device, router rules, or host-based firewall software. These ports frustrate attackers because they provide so little information. Sometimes they respond with ICMP error messages such as type 3 code 13 (destination unreachable) but filters that simply drop probes without responding are far more common. This forces Nmap to retry several times just in case the probe was dropped due to network congestion rather than filtering. This slows down the scan dramatically.
----------	---

Table 7 – Description of Port Scan ‘STATE’ fields.

The daemon must check each time a successful Port Knocking sequence is received if the client is trying to renew a connection. This is done by checking if the client ‘AttemptKnockingSequence’ already has a submitted client-server connection. If so the HashMap value Array List will be cleared and the boolean value for connection renewal will be set.

```
// if renewal connection attempt from client
// clear current connection knock in HashMap of corresponding client AKS
if(aks.getSubmittedConnection() && (hashKnock.get(aks).size() ==
    confirmKnockingSequence.size()) && !aks.getRenewingConnection()) {

    hashKnock.get(aks).clear();
    // set boolean value of renewing connection
    aks.setRenewingConnection(true);
}
```

Next, once the client has sent a Knocking Sequence which fills the HashMap value Array List, the daemon will sort the list of ‘SingleKnock’ based on the NTP timestamp. The list is then verified to be the correct Port Knocking sequence. Once verified, it is checked to see if the client ‘AttemptKnockingSequence’ has a submitted client-server connection, if this connection is a renewal connection or if the connection will be a first for the client and the appropriate action will be taken in each case as shown below. If the incorrect Port Knocking sequence is sent the HashMap value Array List will be cleared.

```
// get hashmap array of single knocks
ArrayList<SingleKnock> arr = hashKnock.get(aks);

// if port knocking array size is now full
if(arr.size() == confirmKnockingSequence.size()) {

    // sort single knocks of client AKS
    // sort based on NTP timestamps in single knock packets
    Collections.sort(hashKnock.get(aks));

    // create knock & connection sequences from HashMap value ArrayList
    ArrayList<Integer> knockSequence = new ArrayList<Integer>();
    ArrayList<Integer> connectionSequence = new ArrayList<Integer>();
    for (SingleKnock sk : hashKnock.get(aks)) {
        knockSequence.add(sk.getPortKnock());
        connectionSequence.add(sk.getConnectionKnock());
    }

    // if knocking sequence matches, client-server connection allowed
    if (knockSequence.equals(confirmKnockingSequence)) {
        // renewing connection attempt
        // but submitted connection has expired
    }
}
```



```

        if(!aks.getSubmittedConnection() &&
            aks.getRenewingConnection()) {

            // new connection with connection knocks
            new Connection(aks, hashKnock.get(aks),
                confirmKnockingPort, logger, hashKnock);
            // boolean values for submitted/renewal connections
            aks.setSubmittedConnection(true);
            aks.setRenewingConnection(false);
        }
        // if new connection attempt
        else if(!aks.getSubmittedConnection()) {
            new Connection(aks, hashKnock.get(aks),
                confirmKnockingPort, logger, hashKnock);
            // set Boolean value for submitted connection
            aks.setSubmittedConnection(true);
            // renewing attempt replaces current submitted connection
        } else {
            aks.setRenewingConnection(false);
        }
        // connection refused due to incorrect incoming knock sequence
        // HashMap ArrayList value cleared of corresponding client AKS
    } else {
        hashKnock.get(aks).clear();
    }
}

```

‘AttemptKnockingKequence’ class getter and setter methods for submitted and renewal client-server connection are shown below. The variables are declared as static due to being accessed by the Connecion.java class and the main method of MyServer.java class. Difficulties arose when calling these methods from both above classes as different boolean values were returned when the static was not declared on the variables. The submitted connection methods are synchronized due to being accessed by both the main method thread and the client ‘AttemptKnockingSequence’ thread.

```

// static variables to check if associated client AKS
// has a submitted or attempting a renewing connection
private static boolean submittedConnection;
private static boolean renewingConnection;

// synchronized getters - accessed by main & submitted connection thread
public synchronized boolean getSubmittedConnection() {
    return submittedConnection;
}

public synchronized void setSubmittedConnection(boolean connection) {
    submittedConnection = connection;
}

public boolean getRenewingConnection() {
    return renewingConnection;
}

public void setRenewingConnection(boolean connection) {
    renewingConnection = connection;
}

```

The ‘compareTo’ method for the ‘SingleKnock’ class is shown below which sorts based on the NTP timestamp.

```
// compare to method to sort single knocks based on NTP timestamp
@Override
public int compareTo(SingleKnock s1) {
    if(Objects.equals(this.ntpTime, s1.ntpTime)) {
        return 0;
    } else if(this.ntpTime > s1.ntpTime) {
        return 1;
    } else {
        return -1;
    }
}
```

The Connection.java class is where the client-server connection occurs. First a 'ScheduledThreadPool' is set up which will allow a single thread to manage the port swapping for each client 'AttemptKnockingSequence'.

```
// each client-server connection is assigned a thread
// Java's 'ScheduledThreadPool' achieves successful threading
private ScheduledExecutorService scheduledThreadPool = null;
this.scheduledThreadPool = Executors.newScheduledThreadPool(1);
```

The client-server connection is started below. The firewall will allow INPUT and OUTPUT UDP and TCP connections from the client IP address to the corresponding Port Knocking sequence port number and vice versa. These firewall rules are run using the 'Runtime' Java library which allows operating system processes to be run. The reason for not using the standard 'ProcessBuilder' here is due to issues arising with some firewall commands not executing when using the library. The 'RunRime' library has a corresponding 'waitFor()' method which ensures the execution the specific command.

```
// initiate client-server connection
// amending INPUT iptables to only accept connections from client ip with
//   marked packed to desired server service on the endPort
// amending OUTPUT iptables to only send back packets to client ip from
//   desired server service on the endPort
public void initiateConnection() {

    String acceptTCPInput = "sudo iptables -A INPUT -p tcp -s " +
        this.address + " --dport " + this.endPort + " -m state -state
        NEW,ESTABLISHED -m mark --mark 1 -j ACCEPT";

    String acceptTCPOutput = "sudo iptables -A OUTPUT -p tcp --sport "
        + this.endPort + " -m state --state ESTABLISHED -j ACCEPT";

    String acceptUDPInput = "sudo iptables -A INPUT -p udp -s " +
        this.address + " --dport " + this.endPort + " -m state -state
        NEW,ESTABLISHED -m mark --mark 1 -j ACCEPT";

    String acceptUDPOutput = "sudo iptables -A OUTPUT -p udp --sport "
        + this.endPort + " -m state --state ESTABLISHED -j ACCEPT";

    // execute above iptables firewall commands
    Runtime.getRuntime().exec(acceptTCPInput).waitFor();
    Runtime.getRuntime().exec(acceptTCPOutput).waitFor();
    Runtime.getRuntime().exec(acceptUDPInput).waitFor();
    Runtime.getRuntime().exec(acceptUDPOutput).waitFor();
}
```

The run method drops the current configured connection port from the IP tables before calling a method to configure the next connection port in the list in the firewall.

```
// run method for client-server connection
// drops MANGLE and NAT PREROUTING iptables commands that have been
// previously amended, then amends new connection port knock
public void run() {

    String dropTCPMangle = "sudo iptables -t mangle -D PREROUTING -p tcp -s " + this.address + " --dport " + this.appendedPort + " -j MARK --set-mark 1";

    String dropTCPNat = "sudo iptables -t nat -D PREROUTING -p tcp -s " + this.address + " --dport " + this.appendedPort + " -j REDIRECT --to-port " + this.endPort;

    String dropUDPMangle = "sudo iptables -t mangle -D PREROUTING -p udp -s " + this.address + " --dport " + this.appendedPort + " -j MARK --set-mark 1";

    String dropUDPNat = "sudo iptables -t nat -D PREROUTING -p udp -s " + this.address + " --dport " + this.appendedPort + " -j REDIRECT --to-port " + this.endPort;

    Runtime.getRuntime().exec(dropTCPMangle).waitFor();
    Runtime.getRuntime().exec(dropTCPNat).waitFor();
    Runtime.getRuntime().exec(dropUDPMangle).waitFor();
    Runtime.getRuntime().exec(dropUDPNat).waitFor();

    // append new connection knock by manipulating iptables
    portAppend();
}
```

If another port connection is not available, the daemon will then check if a client with the same IP has a configured connection. If another client with the same IP has a submitted connection the daemon leaves open the INPUT and OUTPUT commands for the client as closing the commands will cause the client connected to lose their connection. If the client with the same IP has no other connections with the server all INPUT and, OUTPUT firewall rules associated with that client are dropped.

However, if another port connection is available, the appropriate firewall rules will be setup with the next connection port in the HashMap value Array List. The now appended port will be set to the 'appendPort' variable. The client's 'ThreadPool' will then be scheduled to run at fixed intervals initiating the 'run' method which will delete the appended connection port firewall rule and configure the next connection port to be open on the firewall or the client-server connection will be dropped if no more connection ports are left.

```
// run method for client-server connection
// amending MANGLE PREROUTING iptables mark packets from client IP address
// to server connection port
// amending NAT PREROUTING iptables to route packets of client IP address
// from server connection port to server desired service at endPort
public void portAppend() throws InterruptedException {

    List<Integer> connectKnocks = this.getConnectionKnocks();
    int port = 0;
```

```

        // if renewal connection occurs
        // or first-time connection from client IP address and port
        // append first connection knock
        if(!connectKnocks.contains(this.appendedPort)) {
            port = connectKnocks.get(0);
        }
        // if last connection knock already appended, disconnect client
        else if(connectKnocks.indexOf(this.appendedPort) == 3) {
            checkClientDisconnection();
        }
        // else connection knocks left; new connection port retrieved
        else {
            port = connectKnocks.get(
                connectKnocks.indexOf(appendPort) + 1);
        }

        String connectPort = Integer.toString(port);

        String acceptTCPMangle = "sudo iptables -t mangle -A
            PREROUTING -p tcp -s " + this.address + " --dport " +
            connectPort + " -j MARK --set-mark 1";

        String acceptTCPNat = "sudo iptables -t nat -A PREROUTING -p tcp -s
            " + this.address + " --dport " + connectPort + " -j
            REDIRECT --to-port " + this.endPort;

        String acceptUDPMangle = "sudo iptables -t mangle -A PREROUTING -p
            udp -s " + this.address + " --dport " + connectPort + " -j
            MARK --set-mark 1";

        String acceptUDPNat = "sudo iptables -t nat -A PREROUTING -p udp -s
            " + this.address + " --dport " + connectPort + " -j
            REDIRECT --to-port " + this.endPort;

        Runtime.getRuntime().exec(acceptTCPMangle).waitFor();
        Runtime.getRuntime().exec(acceptTCPNat).waitFor();
        Runtime.getRuntime().exec(acceptUDPMangle).waitFor();
        Runtime.getRuntime().exec(acceptUDPNat).waitFor();

        // set new appended port
        this.setAppendedPort(port);
        // schedule thread to run at fixed rate
        // thread filters between disabling and appending connection knocks
        scheduledThreadPool.scheduleAtFixedRate(this, 2, 2,
            TimeUnit.MINUTES);
    }

```

Below is the 'checkClientDisconnection' method. Identifies if a client with the same IP has a connection with the server. If no such connection exists a full client-server disconnection occurs. The method clears the HashMap value Array List of 'SingleKnock' of the corresponding 'AttemptKnockingSequence' client. The 'appendPort' variable is unset meaning no current connection port appended to the firewall and the boolean value for submitted connection is set to false. Finally, the 'ThreadPool' managing the client-server connection is shutdown.

```

logger.info("All connections submitted. Disconnection occurring.");
boolean dropConnection = false;
// synchronize as HashMap has shared access
synchronized(this) {

```

```

        // check if a connection is initialized or connection exists
        // with a client of the same IP so not to fully drop the connection
        for (AttemptKnockingSequence attempt : hm.keySet()) {
            if (attempt.getAddress().equals(this.aks.getAddress()) &&
                attempt.getPort() != this.aks.getPort() &&
                !this.hm.get(new AttemptKnockingSequence(
                    attempt.getAddress(), attempt.getPort())) .isEmpty())
            {
                dropConnection = true;
                break;
            }
        }
    }

    // if a client with the same IP does not currently have a connection
    // fully drop connection with associated client IP address
    if (!dropConnection) {
        dropConnection();
    }

    // set submitted connection to false, reset appended port to -1
    // clear HashMap single knocks of associated attempt knocking sequence
    this.aks.setSubmittedConnection(false);
    this.setAppendedPort(-1);
    synchronized(this) {
        hm.get(aks).clear();
    }

    // shutdown threadpool of current client connection
    this.scheduledThreadPool.shutdown();
    return;
}

```

Deletion of the client firewall rules which fully drops client-server connection.

```

// drops client-server connection
// deleted INPUT & OUTPUT iptables for connections from client IP
private void dropConnection() {

    String dropTCPInput = "sudo iptables -D INPUT -p tcp -s " +
        this.address + " --dport " + this.endPort + " -m state
        --state NEW,ESTABLISHED -m mark --mark 1 -j ACCEPT";

    String dropTCPOutput = "sudo iptables -D OUTPUT -p tcp --sport " +
        this.endPort + " -m state --state ESTABLISHED -j ACCEPT";

    String dropUDPInput = "sudo iptables -D INPUT -p udp -s " +
        this.address + " --dport " + this.endPort + " -m state
        --state NEW,ESTABLISHED -m mark --mark 1 -j ACCEPT";

    String dropUDPOutput = "sudo iptables -D OUTPUT -p udp --sport " +
        this.endPort + " -m state --state ESTABLISHED -j ACCEPT";

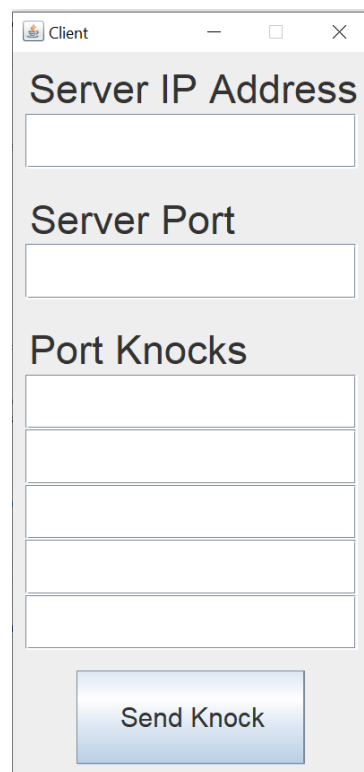
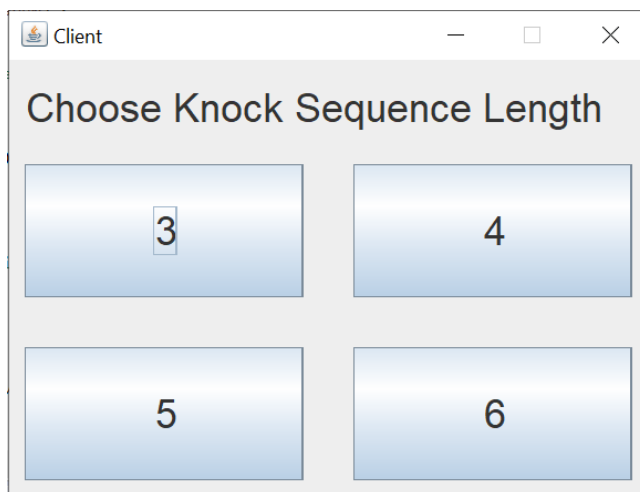
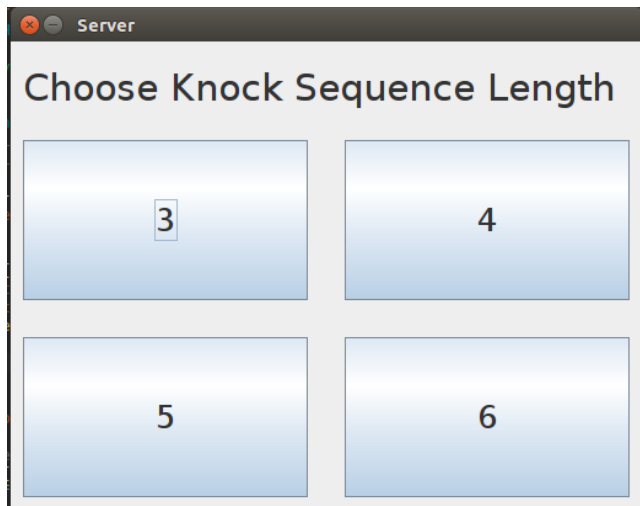
    Runtime.getRuntime().exec(dropTCPInput).waitFor();
    Runtime.getRuntime().exec(dropTCPOutput).waitFor();
    Runtime.getRuntime().exec(dropUDPInput).waitFor();
    Runtime.getRuntime().exec(dropUDPOutput).waitFor();
}

```

This protocol will not use secure VPN tunnelling through IPSec protocol due to the huge overhead to implement such a structure [7].

Graphical User Interface

For application ease of use, a small client and server-side GUI can be run to initiate the Port Knocking connection sequence. The Java Swing framework is used to create the below GUI.



Port Knocking Server via Command Line

Alternatively, a command line option is available to start the **server** using the command –

```
sudo java src.MyServer <port_knocking_sequence_size>
```

The user will then be asked to input the Port Knocking sequence depending on the input size and the server destination port.

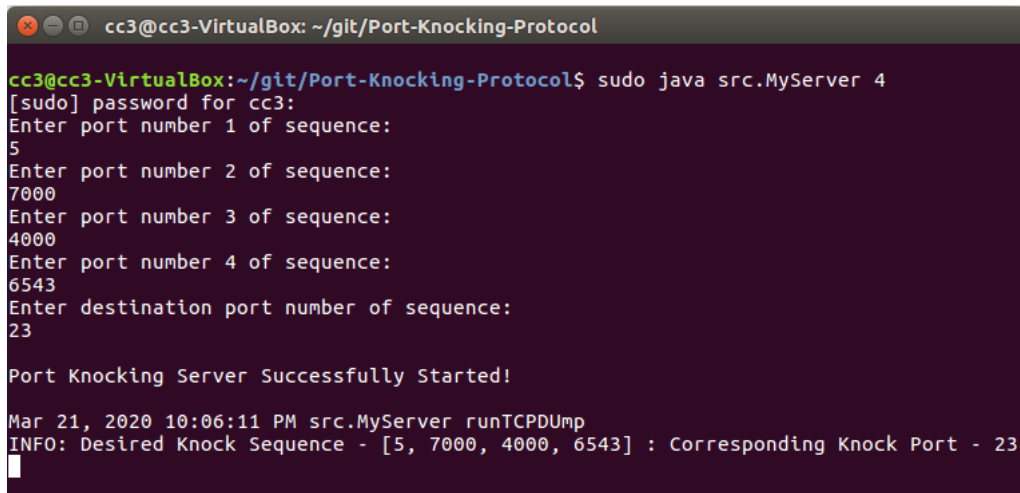
A terminal window titled 'cc3@cc3-VirtualBox: ~/git/Port-Knocking-Protocol'. The user enters the command 'sudo java src.MyServer 4'. The terminal prompts for a password, then asks for the sequence size (4). It then prompts for four port numbers: 5, 7000, 4000, and 6543. Finally, it asks for the destination port (23). After confirmation, it displays 'Port Knocking Server Successfully Started!' and an informational message: 'Mar 21, 2020 10:06:11 PM src.MyServer runTCPDump INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23'.

Figure 31 – Running the Port Knocking server via terminal.

Port Knocking Client via Command Line

To send the Port Knocking sequence from the **client** the following command can be run –

```
sudo java src.MyClient <port_knocking_sequence_size> <server_ip_address>
```

The user will then be asked to input the Port Knocking sequence depending on the input size.

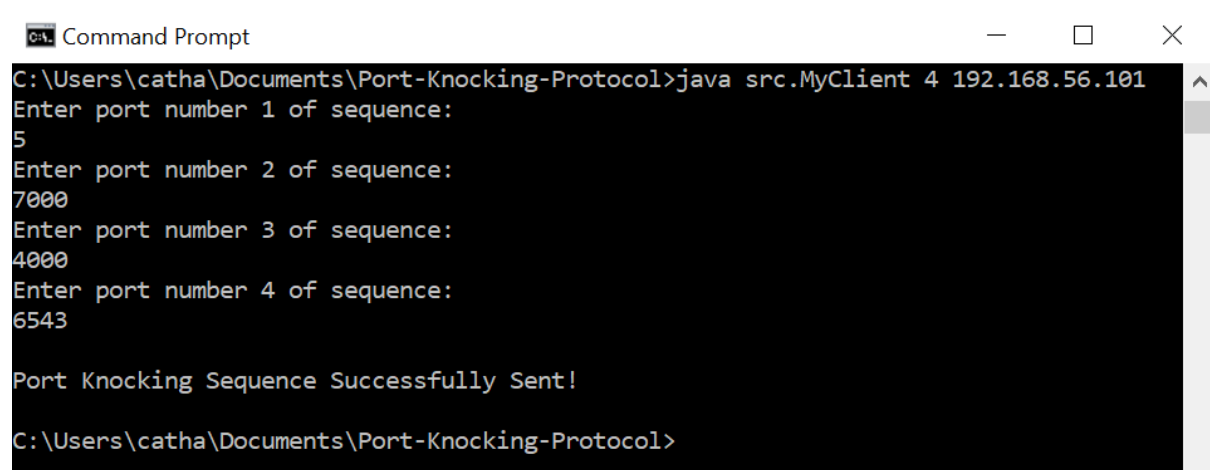
A Windows Command Prompt window titled 'Command Prompt'. The user enters the command 'java src.MyClient 4 192.168.56.101'. The prompt asks for the sequence size (4). It then prompts for four port numbers: 5, 7000, 4000, and 6543. After confirmation, it displays 'Port Knocking Sequence Successfully Sent!' and returns to the command prompt.

Figure 32 – Running the Port Knocking client via command prompt.

Chapter 5 – Testing

Chapter 4 outlines testing performed throughout the Port Knocking Protocol implementation and presents results from the testing stages. A discussion on the results of each testing stage is given.

Note that the Java Port Knocking daemon/server needs to be run with root privileges (as seen from Figure 31). This is due to Linux system such commands as TcpDump and Iptables (discussed later) not being able to run without super user rights

Correct & Fragmented Port Knocking Sequence

The below JUnit test shows a successful Port Knocking client-server connection.

```
@Test
public void testCorrectKnockSequence() {
    ArrayList<Integer> correctKnockingSequence = new
        ArrayList<>(Arrays.asList(5, 7000, 4000, 6543));
    client.sendEcho(correctKnockingSequence);
}
```

The second JUnit test shows a successful Port Knocking client-server connection with the knocks fragmented into different commands.

```
@Test
public void testFragmentedKnockSequence() throws {
    ArrayList<Integer> knockingSequence1 = new
        ArrayList<>(Arrays.asList(5));
    ArrayList<Integer> knockingSequence2 = new
        ArrayList<>(Arrays.asList(7000));
    ArrayList<Integer> knockingSequence3 = new
        ArrayList<>(Arrays.asList(4000, 6543));

    client.sendEcho(knockingSequence1);
    client.sendEcho(knockingSequence2);
    client.sendEcho(knockingSequence3);
}
```

Below shows the log output of a successful Port Knocking sequence attempt and client-server connection of the above two tests.

- Port Knocking Sequence – [5, 7000, 4000, 6543]
- Corresponding Connection Knock Port – 23
- Random Client-Server Connection Ports – [13714, 5628, 47375, 48783]


```

Problems Javadoc Declaration Console x
ServerGui [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (16 Mar 2020, 15:52:57)
Mar 16, 2020 3:53:28 PM src.MyServer runTCPDump
INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 65488: Port Knock Entered - 5
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 65488
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 65488: Port Knock Entered - 7000
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 65488
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 65488: Port Knock Entered - 4000
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 65488
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 65488: Port Knock Entered - 6543
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Correct Knock Sequence: IP - /192.168.56.1: Port - 65488
Mar 16, 2020 3:53:38 PM src.MyServer runTCPDump
INFO: Submitting connection ports for allowed connection: Connection Knocks - [13714, 5628, 47375, 48783]
Mar 16, 2020 3:53:39 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 13714
Mar 16, 2020 3:56:39 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 5628
Mar 16, 2020 3:59:39 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 47375
Mar 16, 2020 3:59:39 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 48783
Mar 16, 2020 4:02:39 PM src.Connection portAppend
INFO: All connections submitted. Disconnection occurring.
Mar 16, 2020 4:02:39 PM src.Connection dropConnection
INFO: Server closing connection with Client IP - /192.168.56.1

```

Figure 33 – Port Knocking daemon log output of successful Port knocking connection.

Realtime screenshot of the firewall rules configured for the first random connection port, 13714, allowing all TCP and UDP connections from the client to server port 13714.

```

cc3@cc3-VirtualBox:~$ sudo iptables -L
[sudo] password for cc3:
Chain INPUT (policy DROP)
target prot opt source destination
ACCEPT udp -- 140.203.204.77 anywhere udp spt:ntp
ACCEPT udp -- localhost localhost
ACCEPT tcp -- 192.168.56.1 anywhere tcp dpt:telnet state NEW,ESTABLISHED mark match 0x1
ACCEPT udp -- 192.168.56.1 anywhere udp dpt:23 state NEW,ESTABLISHED mark match 0x1

Chain FORWARD (policy DROP)
target prot opt source destination

Chain OUTPUT (policy DROP)
target prot opt source destination
ACCEPT udp -- anywhere 140.203.204.77 udp dpt:ntp
ACCEPT udp -- localhost localhost
ACCEPT tcp -- anywhere anywhere tcp spt:telnet state ESTABLISHED
ACCEPT udp -- anywhere anywhere udp spt:23 state ESTABLISHED

cc3@cc3-VirtualBox:~$ sudo -t nat iptables -L
Chain INPUT (policy DROP)
target prot opt source destination
ACCEPT udp -- 140.203.204.77 anywhere udp spt:ntp
ACCEPT udp -- localhost localhost
ACCEPT tcp -- 192.168.56.1 anywhere tcp dpt:telnet state NEW,ESTABLISHED mark match 0x1
ACCEPT udp -- 192.168.56.1 anywhere udp dpt:23 state NEW,ESTABLISHED mark match 0x1

Chain FORWARD (policy DROP)
target prot opt source destination

Chain OUTPUT (policy DROP)
target prot opt source destination
ACCEPT udp -- anywhere 140.203.204.77 udp dpt:ntp
ACCEPT udp -- localhost localhost
ACCEPT tcp -- anywhere anywhere tcp spt:telnet state ESTABLISHED
ACCEPT udp -- anywhere anywhere udp spt:23 state ESTABLISHED

cc3@cc3-VirtualBox:~$ sudo -t mangle iptables -L
Chain INPUT (policy DROP)
target prot opt source destination
ACCEPT udp -- 140.203.204.77 anywhere udp spt:ntp
ACCEPT udp -- localhost localhost
ACCEPT tcp -- 192.168.56.1 anywhere tcp dpt:telnet state NEW,ESTABLISHED mark match 0x1
ACCEPT udp -- 192.168.56.1 anywhere udp dpt:23 state NEW,ESTABLISHED mark match 0x1

Chain FORWARD (policy DROP)
target prot opt source destination

Chain OUTPUT (policy DROP)
target prot opt source destination
ACCEPT udp -- anywhere 140.203.204.77 udp dpt:ntp
ACCEPT udp -- localhost localhost
ACCEPT tcp -- anywhere anywhere tcp spt:telnet state ESTABLISHED
ACCEPT udp -- anywhere anywhere udp spt:23 state ESTABLISHED

```

Figure 34 – Server iptables firewall rules configured for single connection knock.

Below are screenshots from the host machine trying to access the Port Knocking server when the second random connection port (5628) is firewall configured –

- The telnet attempt (default port 23) to the server fails. The server's port 23 is not accepting direct telnet traffic.

- Port scan of port 23 (telnet) shows the traffic is filtered giving no external indication of the port being open.
- Port scan of port 5628 (configured connecting port routing traffic to port 23) shows the port open but no indication that the port is routing telnet traffic.
- Port scan of port 13714 (previous connection port) shows the traffic is filtered showing the port is closed and firewall rules amended to the next connection port (5628).
- Failing telnet connection to the above port confirms that the port is closed.

```

CA Command Prompt

C:\Users\catha>telnet 192.168.56.101
Connecting To 192.168.56.101...Could not open connection to the host, on port 23: Connect failed

C:\Users\catha>nmap 192.168.56.101 -p 23
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-16 15:56 GMT Standard Time
Nmap scan report for 192.168.56.101
Host is up (0.0010s latency).

PORT      STATE      SERVICE
23/tcp    filtered  telnet
MAC Address: 08:00:27:A7:FF:AC (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 1.12 seconds

C:\Users\catha>nmap 192.168.56.101 -p 5628
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-16 15:57 GMT Standard Time
Nmap scan report for 192.168.56.101
Host is up (0.00088s latency).

PORT      STATE      SERVICE
5628/tcp  open       htrust
MAC Address: 08:00:27:A7:FF:AC (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 0.47 seconds

C:\Users\catha>nmap 192.168.56.101 -p 13714
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-16 15:57 GMT Standard Time
Nmap scan report for 192.168.56.101
Host is up (0.00s latency).

PORT      STATE      SERVICE
13714/tcp filtered  netbackup
MAC Address: 08:00:27:A7:FF:AC (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 0.66 seconds

C:\Users\catha>telnet 192.168.56.101 13714
Connecting To 192.168.56.101...Could not open connection to the host, on port 13714: Connect failed

C:\Users\catha>

```

Figure 35 – Port Scan of server port 23, currently configured connection port and previously configured connection port.

Incorrect Port Knocking Sequence

Below is to test sending an incorrect Port Knocking sequence. The daemon's TcpDump command will only capture packets of the confirmed sequence. Therefore, the port 4010 will not be captured by the daemon. The client sleeps for the timeout period of 20 seconds before sending the correct sequence. The daemon recognises the timeout period, clearing the HashMap value Array List and successfully captures the correct sequence and a client-server connection occurs using the random connection ports.

```

@Test
public void test_incorrectKnockSequence_timeout() {
    ArrayList<Integer> incorrectKnockingSequence = new
        ArrayList<>(Arrays.asList(5, 7000, 4010, 6543));
    client.sendEcho(incorrectKnockingSequence);
}

```

```
// timeout period
Thread.sleep(21000);

ArrayList<Integer> correctKnockingSequence = new
    ArrayList<>(Arrays.asList(5, 7000, 4000, 6543));
client.sendEcho(correctKnockingSequence);
}
```

Daemon log output of the incorrect Port Knocking sequence.

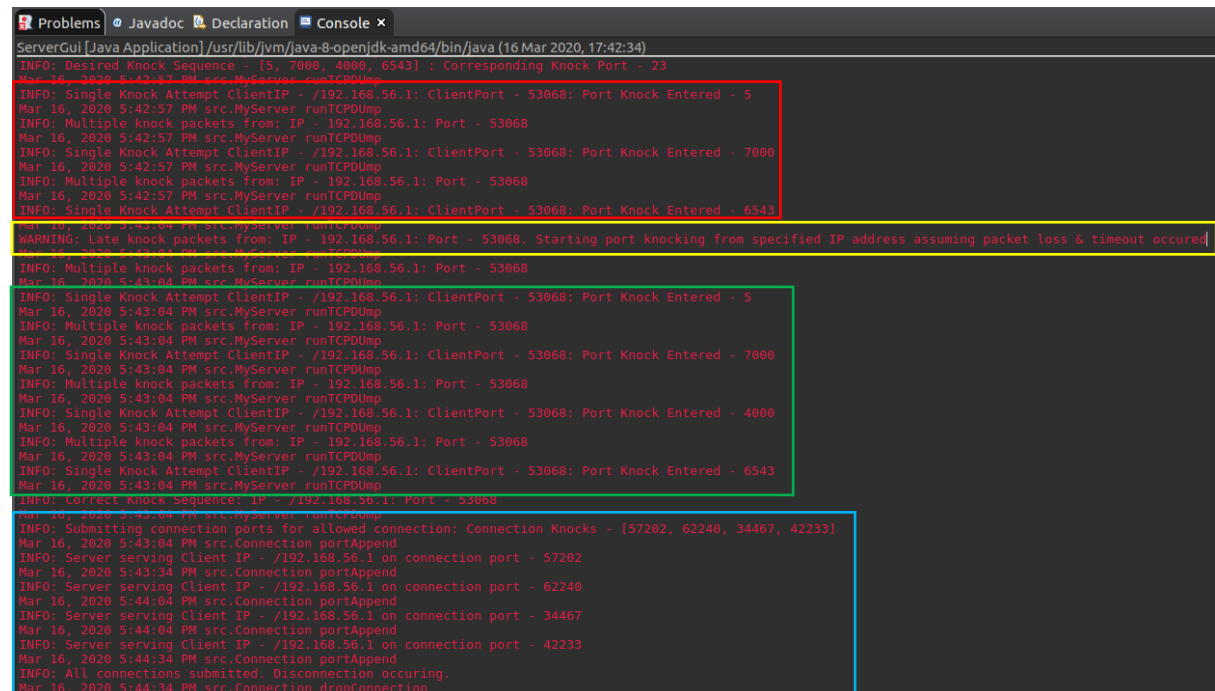


Figure 36 – Port Knocking daemon log output of unsuccessful Port knocking connection followed by successful attempt.

RED	Incorrect Client Port Knocking Sequence.
YELLOW	Daemon Timeout – assumes UDP packet loss. Clear HashMap & restart Port Knocking sequence.
GREEN	Correct Client Port Knocking Sequence.
BLUE	Client-server Connection.

Table 8 – Description of above daemon log output of client Port Knocking sequence.

Late Delivery Packet Port Knocking Sequence

The Java client main method needs to be modified slightly to allow testing of the late delivery packet knocking sequence where the NTP timestamp has been self-modified as seen in the below JUnit test.

```
@Test
public void testLateDeliveryPacket() {
    long time = System.currentTimeMillis();

    Object[] knockingSequenceP1 = {"5", time};
    Object[] knockingSequenceP2 = {"7000", Long.sum(time, 2L)};
    Object[] knockingSequenceP3 = {"4000", Long.sum(time, 4L)};
    Object[] knockingSequenceP4 = {"6543", Long.sum(time, 6L)};

    client.sendEcho(knockingSequence1, portNumber);
    client.sendEcho(knockingSequence2, portNumber);
}
```

```

    client.sendEcho(knockingSequence4, portNumber);
    client.sendEcho(knockingSequence3, portNumber);
}

```

Daemon log output of the late delivery packet of the Port Knocking sequence.

```

ServerGui [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (16 Mar 2020, 19:13:01)
Mar 16, 2020 7:13:18 PM src.MyServer runTCPDump
INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23
Mar 16, 2020 7:13:27 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 63827: Port Knock Entered - 5
Mar 16, 2020 7:13:27 PM src.MyServer runTCPDump
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 63827
Mar 16, 2020 7:13:27 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 63827: Port Knock Entered - 7000
Mar 16, 2020 7:13:27 PM src.MyServer runTCPDump
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 63827
Mar 16, 2020 7:13:27 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 63827: Port Knock Entered - 6543
Mar 16, 2020 7:13:27 PM src.MyServer runTCPDump
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 63827
Mar 16, 2020 7:13:28 PM src.MyServer runTCPDump
INFO: Single Knock Attempt ClientIP - /192.168.56.1: ClientPort - 63827: Port Knock Entered - 4000
Mar 16, 2020 7:13:28 PM src.MyServer runTCPDump
WARNING: Late packet arrival in Single Packet Knock: IP - /192.168.56.1: Port - 63827
Mar 16, 2020 7:13:28 PM src.MyServer runTCPDump
INFO: Correct Knock Sequence: IP - /192.168.56.1: Port - 63827
Mar 16, 2020 7:13:28 PM src.MyServer runTCPDump
INFO: Submitting connection ports for allowed connection: Connection Knocks - [18367, 156, 32392, 63737]
Mar 16, 2020 7:13:28 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 18367
Mar 16, 2020 7:13:58 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 156
Mar 16, 2020 7:14:28 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 32392
Mar 16, 2020 7:14:29 PM src.Connection portAppend
INFO: Server serving Client IP - /192.168.56.1 on connection port - 63737
Mar 16, 2020 7:14:58 PM src.Connection portAppend
INFO: All connections submitted. Disconnection occurring.
Mar 16, 2020 7:14:58 PM src.Connection dropConnection
INFO: Server closing connection with Client IP - /192.168.56.1

```

Figure 37 – Port Knocking daemon log output of UDP late packet arrival.

Time Modification Port Knocking Sequence

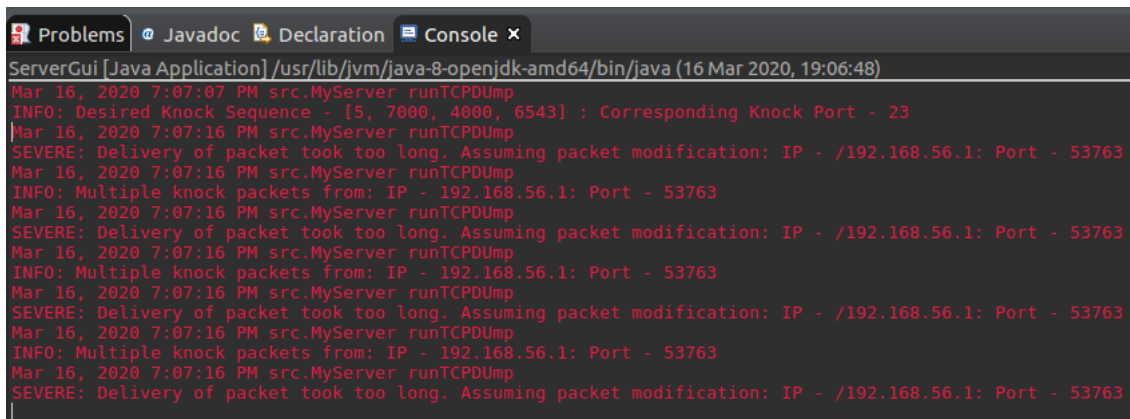
An attacker may capture a previous Port Knocking sequence sent by a valid client and replay packets to access the server.

```

@Test
public void testTimePacketModification () {
    Object[] knockingSequence1 = {"5", 123456781L};
    Object[] knockingSequence2 = {"7000", 123456785L};
    Object[] knockingSequence3 = {"4000", 123456787L};
    Object[] knockingSequence4 = {"6543", 123456789L};

    client.sendEcho(knockingSequence1, portNumber);
    client.sendEcho(knockingSequence2, portNumber);
    client.sendEcho(knockingSequence4, portNumber);
    client.sendEcho(knockingSequence3, portNumber);
}

```

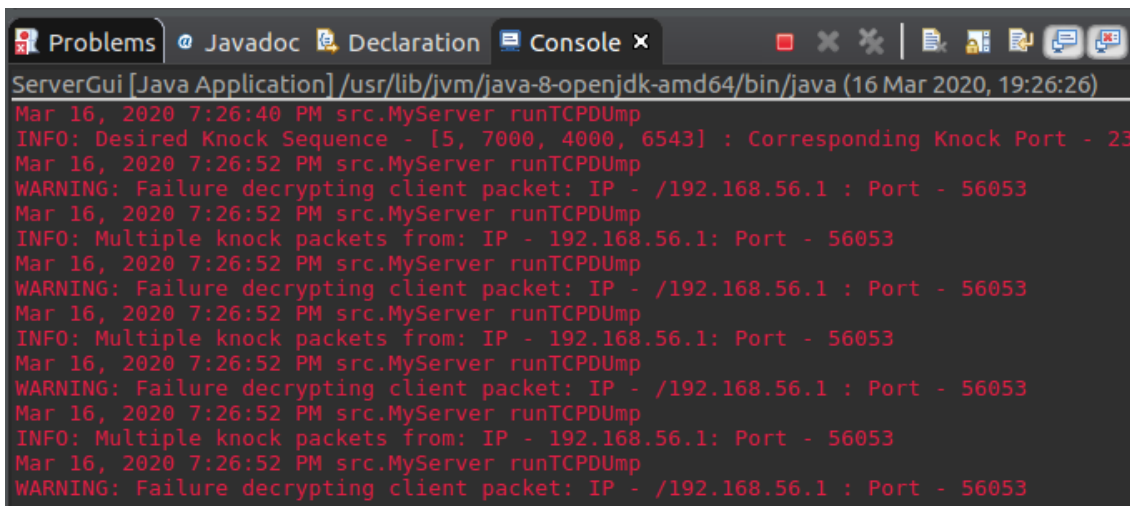


```
Problems | Javadoc | Declaration | Console x
ServerGui [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (16 Mar 2020, 19:06:48)
Mar 16, 2020 7:07:07 PM src.MyServer runTCPDUMP
INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23
Mar 16, 2020 7:07:16 PM src.MyServer runTCPDUMP
SEVERE: Delivery of packet took too long. Assuming packet modification: IP - /192.168.56.1: Port - 53763
Mar 16, 2020 7:07:16 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 53763
Mar 16, 2020 7:07:16 PM src.MyServer runTCPDUMP
SEVERE: Delivery of packet took too long. Assuming packet modification: IP - /192.168.56.1: Port - 53763
Mar 16, 2020 7:07:16 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 53763
Mar 16, 2020 7:07:16 PM src.MyServer runTCPDUMP
SEVERE: Delivery of packet took too long. Assuming packet modification: IP - /192.168.56.1: Port - 53763
Mar 16, 2020 7:07:16 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 53763
Mar 16, 2020 7:07:16 PM src.MyServer runTCPDUMP
SEVERE: Delivery of packet took too long. Assuming packet modification: IP - /192.168.56.1: Port - 53763
```

Figure 38 – Port Knocking daemon log output of time modification of UDP packet.

Without Encryption Port Knocking Sequence

Daemon log output of the failure to encrypt using a valid server public key.

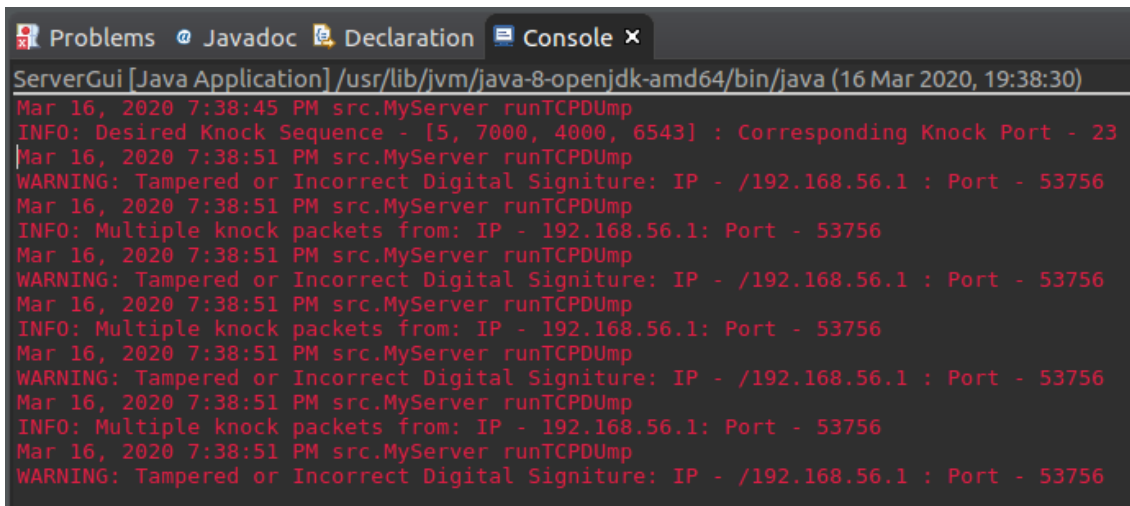


```
Problems | Javadoc | Declaration | Console x
ServerGui [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (16 Mar 2020, 19:26:26)
Mar 16, 2020 7:26:40 PM src.MyServer runTCPDUMP
INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23
Mar 16, 2020 7:26:52 PM src.MyServer runTCPDUMP
WARNING: Failure decrypting client packet: IP - /192.168.56.1 : Port - 56053
Mar 16, 2020 7:26:52 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 56053
Mar 16, 2020 7:26:52 PM src.MyServer runTCPDUMP
WARNING: Failure decrypting client packet: IP - /192.168.56.1 : Port - 56053
Mar 16, 2020 7:26:52 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 56053
Mar 16, 2020 7:26:52 PM src.MyServer runTCPDUMP
WARNING: Failure decrypting client packet: IP - /192.168.56.1 : Port - 56053
Mar 16, 2020 7:26:52 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 56053
Mar 16, 2020 7:26:52 PM src.MyServer runTCPDUMP
WARNING: Failure decrypting client packet: IP - /192.168.56.1 : Port - 56053
```

Figure 39 – Port Knocking daemon log output of unencrypted Port knocking sequence.

Incorrect or Tampered Digital Signature Port Knocking Sequence

Daemon log output of an incorrect digital signature or a data packet that has been signed with an invalid signature.

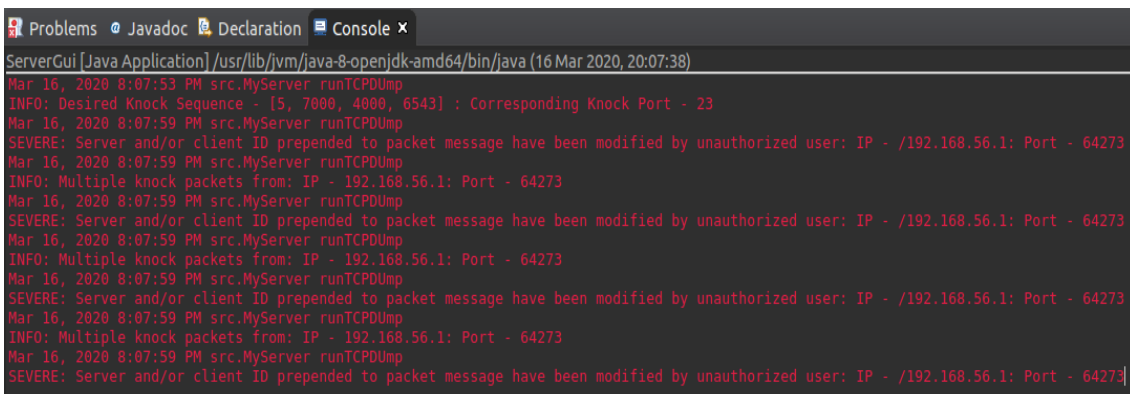


```
Problems Javadoc Declaration Console x
ServerGui [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (16 Mar 2020, 19:38:30)
Mar 16, 2020 7:38:45 PM src.MyServer runTCPDUMP
INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23
Mar 16, 2020 7:38:51 PM src.MyServer runTCPDUMP
WARNING: Tampered or Incorrect Digital Signature: IP - /192.168.56.1 : Port - 53756
Mar 16, 2020 7:38:51 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 53756
Mar 16, 2020 7:38:51 PM src.MyServer runTCPDUMP
WARNING: Tampered or Incorrect Digital Signature: IP - /192.168.56.1 : Port - 53756
Mar 16, 2020 7:38:51 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 53756
Mar 16, 2020 7:38:51 PM src.MyServer runTCPDUMP
WARNING: Tampered or Incorrect Digital Signature: IP - /192.168.56.1 : Port - 53756
Mar 16, 2020 7:38:51 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 53756
Mar 16, 2020 7:38:51 PM src.MyServer runTCPDUMP
WARNING: Tampered or Incorrect Digital Signature: IP - /192.168.56.1 : Port - 53756
```

Figure 40 – Port Knocking daemon log output of Port Knocking attempt of tampered with digital signature.

Falsely Prepended Client/Server ID in Packet Data

The client or server IP address appended to the plaintext string in the data packet does not match the client that has sent signed the packet indicating the packet has been tampered with or modified.



```
Problems Javadoc Declaration Console x
ServerGui [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (16 Mar 2020, 20:07:38)
Mar 16, 2020 8:07:53 PM src.MyServer runTCPDUMP
INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23
Mar 16, 2020 8:07:59 PM src.MyServer runTCPDUMP
SEVERE: Server and/or client ID prepended to packet message have been modified by unauthorized user: IP - /192.168.56.1: Port - 64273
Mar 16, 2020 8:07:59 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 64273
Mar 16, 2020 8:07:59 PM src.MyServer runTCPDUMP
SEVERE: Server and/or client ID prepended to packet message have been modified by unauthorized user: IP - /192.168.56.1: Port - 64273
Mar 16, 2020 8:07:59 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 64273
Mar 16, 2020 8:07:59 PM src.MyServer runTCPDUMP
SEVERE: Server and/or client ID prepended to packet message have been modified by unauthorized user: IP - /192.168.56.1: Port - 64273
Mar 16, 2020 8:07:59 PM src.MyServer runTCPDUMP
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 64273
Mar 16, 2020 8:07:59 PM src.MyServer runTCPDUMP
SEVERE: Server and/or client ID prepended to packet message have been modified by unauthorized user: IP - /192.168.56.1: Port - 64273
```

Figure 41 – Port Knocking daemon log output of falsely appended client/server IP address.

Client Public Key for Signature Verification Not Shared

Where the client has not shared their public key with the sever or the server's client public key management tool does not contain an associated public key for the incoming client.


```

ServerGui [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (16 Mar 2020, 20:14:14)
Mar 16, 2020 8:14:28 PM src.MyServer runTCPDUmp
INFO: Desired Knock Sequence - [5, 7000, 4000, 6543] : Corresponding Knock Port - 23
Mar 16, 2020 8:14:48 PM src.MyServer runTCPDUmp
WARNING: Server does not contain a key for client: IP - /192.168.56.1 : Port - 59591
Mar 16, 2020 8:14:48 PM src.MyServer runTCPDUmp
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 59591
Mar 16, 2020 8:14:48 PM src.MyServer runTCPDUmp
WARNING: Server does not contain a key for client: IP - /192.168.56.1 : Port - 59591
Mar 16, 2020 8:14:48 PM src.MyServer runTCPDUmp
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 59591
Mar 16, 2020 8:14:48 PM src.MyServer runTCPDUmp
WARNING: Server does not contain a key for client: IP - /192.168.56.1 : Port - 59591
Mar 16, 2020 8:14:48 PM src.MyServer runTCPDUmp
INFO: Multiple knock packets from: IP - 192.168.56.1: Port - 59591
Mar 16, 2020 8:14:48 PM src.MyServer runTCPDUmp
WARNING: Server does not contain a key for client: IP - /192.168.56.1 : Port - 59591

```

Figure 42 – Port Knocking daemon log output when the server contains no associated client public key.

Encryption & Decryption Testing

Below is a test comparing AES-RSA hybrid against RSA only asymmetric encryption and decryption with digital signature.

	RSA & AES w/ Digital Signature		RSA Only w/ Digital Signature	
	Encryption	Decryption	Encryption	Decryption
Test 1 (milliseconds)	427	413	402	309
	17	51	18	88
	22	31	16	137
	27	31	29	124
	22	18	23	28
	13	23	20	22
Test 2 (milliseconds)	10	395	7	44
	8	50	7	24
	7	29	7	32
	7	46	5	33
	5	19	7	60
	6	10	7	38
Test 3 (milliseconds)	15	22	6	17
	8	17	6	31
	7	12	5	26
	8	14	6	13
	5	46	4	19
	5	13	9	16
Test 4 (milliseconds)	9	13	9	8
	8	8	7	7
	8	8	8	13
	7	13	6	18
	6	5	8	5
	6	6	8	10

Table 9 – Test results of Hybrid vs Asymmetric encryption and decryption.

Below are corresponding graphs comparing the above encryption and decryption times for the two cryptographic standards. The first encryption and decryption time for each 'Test 1' has been ignored due to the protocol outputting large compute times when the first packet of the first sequence is sent.

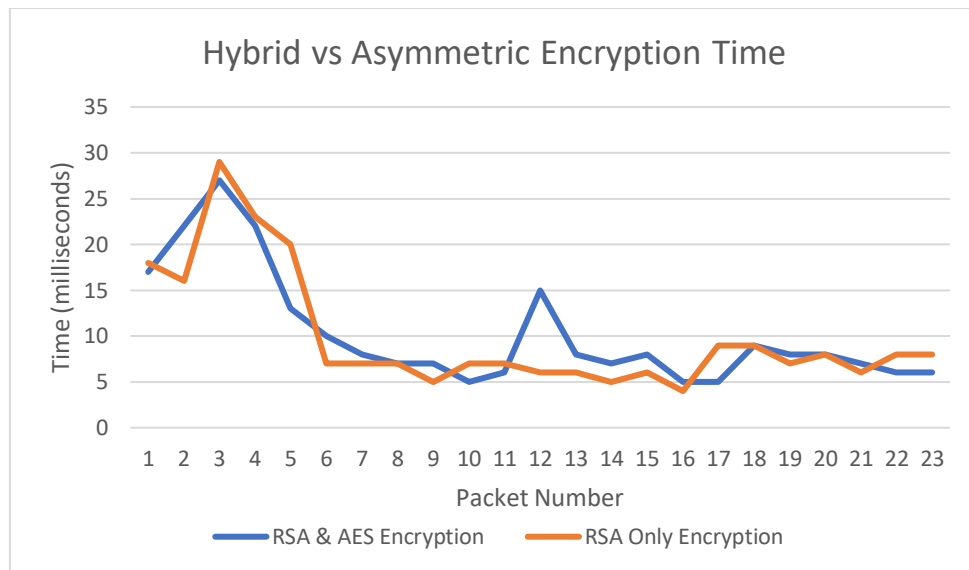


Figure 43 – Graph of Hybrid vs Asymmetric Packet Encryption Time.

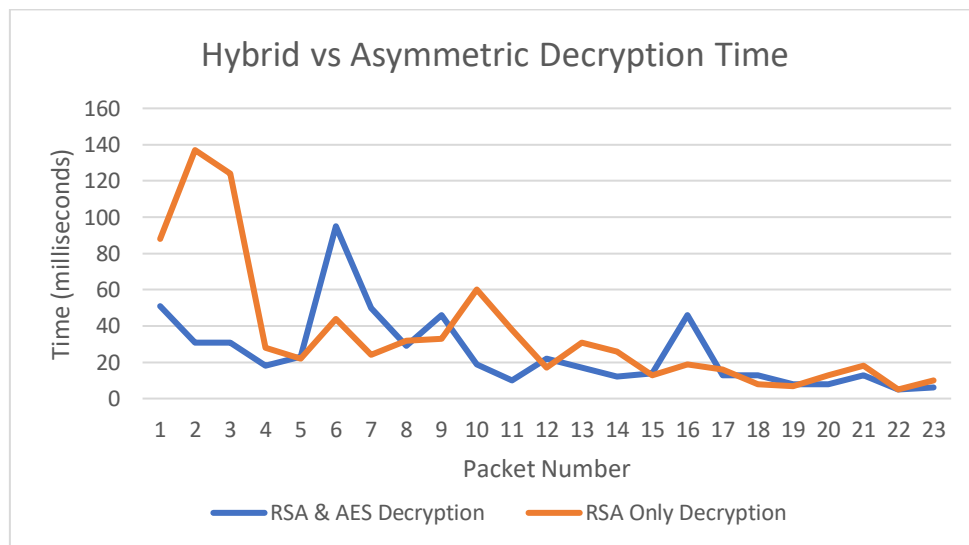
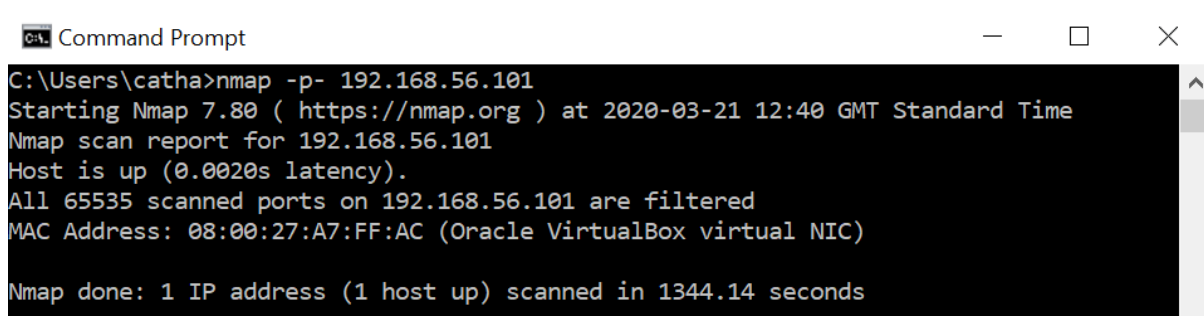


Figure 44 – Graph of Hybrid vs Asymmetric Decryption Time.

As seen from the graphs, both encryption and decryption times are very similar. The first few Port Knocking sequences in both instances can experience large variations in compute time. However, encryption and decryption times for both cryptographic standards level off after some time and become steady.

Port Scan Test

Port scan of all TCP ports which takes 1344 seconds or just over 22 minutes.

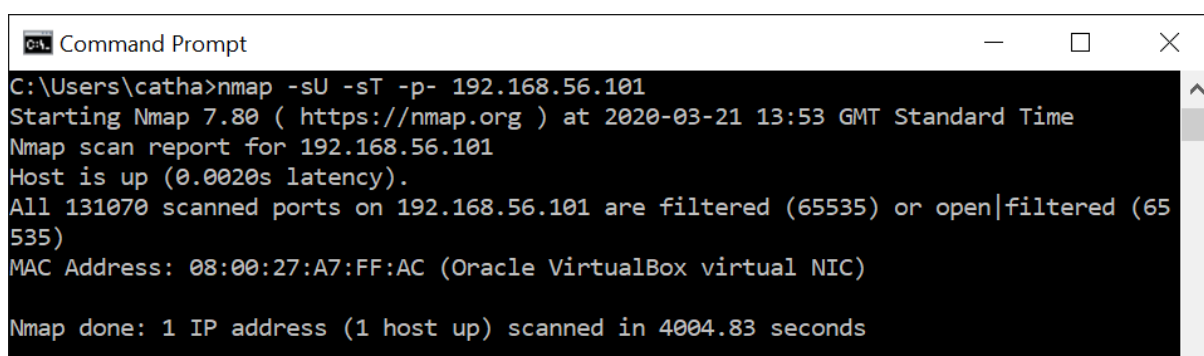


```
C:\Users\catha>nmap -p- 192.168.56.101
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-21 12:40 GMT Standard Time
Nmap scan report for 192.168.56.101
Host is up (0.0020s latency).
All 65535 scanned ports on 192.168.56.101 are filtered
MAC Address: 08:00:27:A7:FF:AC (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 1344.14 seconds
```

Figure 45 – Time taken to port scan all TCP server ports.

Port scan of all TCP and UDP ports which takes 4005 seconds or just over 66 minutes.



```
C:\Users\catha>nmap -sU -sT -p- 192.168.56.101
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-21 13:53 GMT Standard Time
Nmap scan report for 192.168.56.101
Host is up (0.0020s latency).
All 131070 scanned ports on 192.168.56.101 are filtered (65535) or open|filtered (65535)
MAC Address: 08:00:27:A7:FF:AC (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 4004.83 seconds
```

Figure 46 – Time taken to port scan all TCP and UDP server ports.

The above times represent how long it would take an attacker to Port Scan a computer they would like to compromise. The above times shows how useless it would be for an attacker to brute force the above protocol to check for altered port action. If a connection port for each Port Knocking sequence was configured in the server firewall for 20 minutes, an attacker would have failed to complete a Port Scan of just TCP ports which takes approximately 22 minutes.

Chapter 6 – Conclusion

Chapter 5 evaluates the protocol comparing the results to the introductory outline scope of the project. A discussion on aspects of the project that could be improved or investigated to further advance work of the Port Knocking Protocol will be given in the future work segment.

The Port Knocking Protocol implemented is secure and highly authenticated, leaving little room for an attacker to compromise the application or exploit the Port Knocking connection port through port scanning. However, the above features come at a cost.

Testing shows the Hybrid Encryption and Asymmetric Encryption with Digital Signature performing somewhat similar when the protocol stabilises after some time after encrypting and decrypting a few packets. However, the Hybrid Encryption lends itself to be more secure than the Asymmetric alternative as it naturally solves all the downfalls that exist with the defective ‘Encrypt-Then-Sign’ procedure when applying Digital Signature to the encrypted message. The ‘Encrypt-Then-Sign’ method also has a huge advantage of verifying the signature first which defends against brute-force attacks rather than decrypting all incoming packets first which would compromise the system due to the time intensive task. This would also be a far better method when applied to IoT devices due to their limited processing power which would be easily fully utilized in a DOS attack if the device has to decrypt packet contents first before verifying an unauthorized signature.

The above protocol performs best if a larger Port Knocking sequence is used (6 packet sequence) with less time for each connection port (15 – 20 minutes per connection port). Therefore, an attacker would have larger combination of brute force attempts to try and obtain the successful sequence. If perhaps the attacker guesses the correct sequence, performing port scanning within a 15 to 20-minute period to retrieve the associated open port is highly unlikely as to scan all UDP and TCP ports from Figure 46 takes over 60 minutes.

Additionally, being able to hide the desired connection port is a huge advantage to defend against attackers as they are unaware what port is running the valuable application. This allows the application to be successfully run without any attacker disruptions. Only the running service will need to be agreed with the client and the desired port number running the service should not need to be shared with the client as client-server connection occurs through random packet encrypted connection knocks. This is advantageous when the server shares the Port Knocking sequence with the client, as an attacker who could potentially intercept the sequence would not be able identify the corresponding connection port. In reality, IoT devices are unmonitored for the most part and log files of malicious attacks are often not read by users who could potentially amend the device firewall to keep malicious IP addresses out. Therefore, being able to hide the port of an unmonitored prevents an attacker doing port scanning to reveal and exploit what port the service is running on.

Exception handling has been performed extremely successfully on the protocol as shown through the above tests ensuring the continuous running of the Port Knocking daemon.

For each client who wishes to gain access to the server a public key management tool much exist to hold each client public key to verify digital signatures. The Port Knocking protocol implemented uses a HashMap to store the public keys for testing purposes. However, a proper key management tool should be used to allow the daemon manage client public keys securely. Unfortunately, the storage of each client public key is potentially a large storage overhead for IoT devices.

The implemented protocol assumes that the 'secret' Port Knocking sequence has been previously agreed on by the client and server in a secure manner. However, in a real-world application the client and server will have to agree on the Port Knocking sequence in a secure way between one another.

The proposed protocol assumes that all server ports can be used for client-server connection when choosing the random connection ports. This method is sufficient if the server is running specific applications on specific ports with no well-known port ranges being used. However, in a real-world application this cannot be the case. Well-known ports (such as ports 0 – 1024) are assigned ports to specific applications and protocols running. Depending on the server requirements, if a server is running applications on this port range or other well-known port ranges, the protocol cannot use these port ranges as connection attempts to the server. It would be the server's responsibility to notify the client of the ports which the client-server connection would not be allowed. This would potentially reduce the security of the protocol as an attacker with knowledge of the well-known port ranges would have less of a port range to port scan.

Future Work

The key exchange problem has not been solved within this application. Having asymmetric keys for both client (authentication) and server (encryption) and transferring them securely to both machines is a large overhead to solve for such an application. The asymmetric key exchange is a huge weakness in the above protocol. Without the ability of the client and server to exchange their keys in an appropriate manner, the Port Knocking protocol cannot be initiated. For future work, being able to implement an appropriate key exchange mechanism to facilitate the protocol would make a complete application.

Secondly, finding a secure way to share secret Port Knocking sequence between the client and server is essential. The above application assumes the 'secret' sequence is known by both entities prior to starting the application.

To make a robust application it would be useful to test other asymmetric algorithms such as Elliptic Curve Cryptography and compare against the above RSA algorithm. It is important for the daemon to be able to decrypt packet contents as efficiently as possible to defend against brute-force attacks. Further testing will also need to be done into hybrid encryption with other asymmetric applications to verify if the application works as well using the hybrid approach.

Because NAT exists to overcome a shortage of IPv4 addresses, and because IPv6 has no such shortage, IPv6 networks do not require NAT. Therefore, with majority of devices these days configured to use IPv6, the protocol should use IPv6 over IPv4 due to the NAT attack vulnerability that can exist with the Port Knocking protocol. However, investigating more into applying the protocol to IPv6 more would be required to reduce the possibility of NAT attacks and influence user to use IPv6 over the traditional IPv4. The protocol had intended to implement Port Knocking over IPv6 for the extra protection of IoT devices from NAT attacks. However, problems existed with home broadband and phone hotspot connection and both ISP's not having IPv6 connection enabled.

Finally, the above protocol has a large reliance on asymmetric keys, encryption and digital signatures to ensure a secure and authenticated protocol. Much of this reliance could be replaced by implementing Digital Certificates and use an established Certificate Authority to manage certificate exchange. This would not mean that Digital Certs would be the right decision for the above protocol but would be an investigation to see if it could improve the protocol.

References

- [1] Wikipedia, "Port Knocking," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Port_knocking. [Accessed 03 11 2019].
- [2] R. . deGraaf, J. . Aycok and M. . Jacobson, "Improved port knocking with strong authentication," , 2005. [Online]. Available: <https://acsac.org/2005/papers/156.pdf>. [Accessed 3 11 2019].
- [3] T. Popeea, V. Olteanu, L. Gheorghe and R. Rughinis, "Extension of a port knocking client-server architecture with NTP synchronization," in *10th Roedunet International Conference (RoEduNet)*, Iasi, Romania, 2011.
- [4] D. Sel, S. H. Totakura and G. Carle, "sKnock: Port-Knocking for Masses," in *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, 2016.
- [5] Wikipedia, "Security Through Obscurity," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Security_through_obscurity. [Accessed 01 03 2020].
- [6] F. H. M. Ali, R. . Yunos and M. A. M. Alias, "Simple port knocking method: Against TCP replay attack and port scanning," , 2012. [Online]. Available: <https://ieeexplore.ieee.org/document/6246118>. [Accessed 3 11 2019].
- [7] J.-H. . Liew, S. . Lee, I. . Ong, H.-J. . Lee and H. . Lim, "One-Time Knocking framework using SPA and IPsec," , 2010. [Online]. Available: <http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.ieee-000005529780>. [Accessed 3 11 2019].
- [8] H. Al-Bahadili and A. H. Hadi, "Network Security Using Hybrid Port Knocking," in *2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, Udaipur, India, 2011.
- [9] V. . Srivastava, A. K. Keshri, A. D. Roy, V. K. Chaurasiya and R. . Gupta, "Advanced port knocking authentication scheme with QRC using AES," , 2011. [Online]. Available: <https://ieeexplore.ieee.org/document/5958506>. [Accessed 3 11 2019].
- [10] K. Mariam, A. Hadi and A. Hudaib, "Covert Communication Using Port Knocking," in *IEEE Software, Computer Science, Princess Sumaya University of Technology Amman*, Jordan, 2016.
- [11] Wikipedia, "Network Time Protocol," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Network_Time_Protocol. [Accessed 07 11 2019].
- [12] B. Mahbooba and M. Schukat, "Digital certificate-based port knocking for connected embedded systems," in *2017 28th Irish Signals and Systems Conference (ISSC)*, Killarney, Ireland, 2017.
- [13] P. . Mehran, E. A. Reza and B. . Laleh, "SPKT: Secure Port Knock-Tunneling, an enhanced port security authentication mechanism," *Information Sciences*, vol. , no. , pp. 145-149, 2012.

- [14] B. Nicolau, "VirtualBox: How to set up networking so both host and guest can access internet and talk to each other," StackExchange, [Online]. Available: <https://serverfault.com/questions/225155/virtualbox-how-to-set-up-networking-so-both-host-and-guest-can-access-internet>. [Accessed 07 01 2020].
- [15] L. Wagner, "A Very Basic Introduction to AES-256 Cipher," Hackernoon, 17 01 2020. [Online]. Available: <https://hackernoon.com/very-basic-intro-to-aes-256-cipher-qxr32yk>. [Accessed 30 03 2020].
- [16] B. a. S. R. K. Padmavathi, *A Survey on Performance Analysis of DES, AES and RSA Algorithm along with LSB Substitution Technique.*, 2013.
- [17] Zend Framework, "Hybrid Cryptosystem," Zend Framework, [Online]. Available: <https://docs.zendframework.com/zend-crypt/hybrid/>. [Accessed 17 03 2020].
- [18] T. Schmidt, "The digital signature – your electronic signature," EASY NEWSROOM, 04 11 2019. [Online]. Available: <https://easy-software.com/en/newsroom/the-digital-signature-your-electronic-signature/>. [Accessed 21 03 2020].
- [19] D. Davis, "Defective Sign & Encrypt in S/MIME,," 05 05 2001. [Online]. Available: http://world.std.com/~dtd/sign_encrypt/sign_encrypt7.html. [Accessed 21 01 2020].
- [20] M. Panagiotidou, "Java – Hybrid Cryptography example," Mkyong.com, 30 11 2016. [Online]. Available: <https://mkyong.com/java/java-hybrid-cryptography-example/>. [Accessed 16 01 2020].
- [21] NTP Pool Project, "Ireland — ie.pool.ntp.org," NTP Pool Project, [Online]. Available: <https://www.pool.ntp.org/zone/ie>. [Accessed 05 02 2020].
- [22] Masterclock, "Network Synchronization: Public NTP Servers vs. GPS NTP Servers," Masterclock, 29 03 2018. [Online]. Available: <https://www.masterclock.com/company/masterclock-inc-blog/network-synchronization-internet-ntp-servers-vs-gps-ntp-servers>. [Accessed 03 02 2020].
- [23] jpcap, "jpcap -- a network packet capture library for applications written in Java,," jpcap, [Online]. Available: <http://jpcap.sourceforge.net/>. [Accessed 28 11 2019].
- [24] K. Yamada, "Pcap4J A Java library for capturing, crafting, and sending packets,," Pcap4J, [Online]. Available: <https://www.pcap4j.org/>. [Accessed 31 10 2019].
- [25] man2html, "Manpage of TCPDUMP," man2html, [Online]. Available: <https://www.tcpdump.org/manpages/tcpdump.1.html>. [Accessed 02 03 2020].
- [26] TheHelix, "Redirect traffic between two ports with iptables," StackExchange, [Online]. Available: <https://serverfault.com/a/386522>. [Accessed 14 02 2020].
- [27] NMAP.ORG, "Port Scanning Basics," nmap.org, [Online]. Available: <https://nmap.org/book/man-port-scanning-basics.html>. [Accessed 02 01 2020].

- [28] C. Goedegebure, "TCP 3-way handshake and port scanning," 18 09 2018. [Online]. Available: <https://www.coengoedegebure.com/tcp-3-way-handshake-port-scanning/>. [Accessed 31 03 2020].